University of Helsinki Report Series in Physics

SPINFUL ALGORITHMIZATION OF HIGH ENERGY DIFFRACTION

Mikael Mieskolainen

Department of Physics, Faculty of Science

DOCTORAL DISSERTATION

To be presented for public discussion with the permission of the Faculty of Science of the University of Helsinki, in the Lars Ahlfors Auditorium A111, Exactum, on 20th of March, 2020, at 12 o'clock.

Mentor
Professor Emeritus Risto Orava
University of Helsinki, Finland

Reviewers
Professor Tuomas Lappi
University of Jyväskylä, Finland

Professor David Milstead
Stockholm University, Sweden

Opponent
Professor Valery Khoze
Institute for Particle Physics Phenomenology
University of Durham, UK

Officer
Professor Kenneth Österberg
University of Helsinki, Finland

ISSN 0356-0961
Report Series in Physics HU-P-D270

ISBN 978-951-51-5942-7 (print)
ISBN 978-951-51-5943-4 (pdf)
https://ethesis.helsinki.fi
Unigrafia Oy

Helsinki 2020

Mikael Mieskolainen:
SPINFUL ALGORITHMIZATION OF HIGH ENERGY DIFFRACTION
407 pp, University of Helsinki, Report Series in Physics, 2020.

Abstract

High energy diffraction probes fundamental interactions, the vacuum, and quantum mechanically coherent matter waves at asymptotic energies. In this work, we algorithmize our abstract ideas and develop a set of rigid rules for diffraction. To get spin under control, we construct a new Monte Carlo simulation engine, GRANIITTI. It is the first event generator with custom spin-dependent scattering amplitudes for the glueball domain semi-exclusive diffraction, driven by fully multithreaded importance sampling and written in C++. Our simulations provide new computational evidence that the enigmatic glueball filter observable is a spin polarization filter for tensor resonances. For algorithmic spin studies, we automate the classic Laplace spherical harmonics inverse expansion, carefully define the geometric acceptance related phase space issues and study the harmonic mixing properties systematically in different Lorentz frames.

To improve the big picture, we generalize the standard soft diffraction observables and definitions by developing a high dimensional probabilistic framework based on incidence algebras, COMBINATORIAL SUPERSTATISTICS, and solve also a new superposition inverse problem using the Möbius inversion theorem. For inverting stochastic autoconvolution integral equations or 'inverting the proton', we develop a novel recursive inverse algorithm based on the Fast Fourier Transform and relative entropy minimization. The first algorithmic inverse results of the proton double multiplicity structure and multiparton interaction rates are obtained using the published LHC data, in agreement with standard phenomenology. For optimal inversion of the detector efficiency response, we build the first Deep Learning based solution working in higher phase space dimensions, DEEPEFFICIENCY, which inverts the detector response on an event-by-event basis and minimizes the event generator dependence.

Using the ALICE experiment proton-proton data at the LHC at $\sqrt{s} = 13$ TeV, we obtain the first unfolded fiducial measurement of the multidimensional combinatorial partial cross sections, the first multidimensional maximum likelihood fit of the effective soft pomeron intercept and the first multidimensional maximum likelihood fit of the single, double and non-diffractive component cross sections. Great care is taken with the fiducial and non-fiducial definitions. The second topic of measurements centers on semi-exclusive central diffractive production of hadron pairs, which we study with the ALICE data. We measure and fit the resonance spectra of identified pion and kaon pairs, which is crucial on the road towards solving the mysteries of glueballs, the proton structure fluctuations, and the pomeron.

Acknowledgements

This thesis is the result of 5 years of work at CERN and in Helsinki. I would like to thank Professor Orava for a great project and giving me freedom to independently attack many cutting edge physics problems. Some problems were solved, some new were found out. My skills were also greatly enhanced during the years by being able to supervise numerous summer students and trainees at CERN and on teaching the fundamentals of particle physics on the demanding crash course IPP1 in Helsinki.

I would like to thank Professor Khoze for being the opponent in the public defence, Professors Lappi and Milstead for their reviews and comments on the work, and Professor Österberg for being the officer for this thesis.

For useful discussions or remarks during the project, I would like to thank the following persons: R. Orava for several years of discussions, and in an alphabetical order: M. Albrow, E. Brücken, P. Buehler, S.U. Chung, R. Ciesielski, S. Evdokimov, L. Goerlich, K. Golec-Biernat, C. Gütschow, H. Hirvonsalo, L. Jenkovszky, T. Kim, K. Kowalski, E. Kryshen, P. Kupiainen, J. Lämsä, J. Lång, T. Mäkelä, J. Nystrand, S. Patomäki, J. Pekkanen, J. Pinfold, V. Pyykkönen, S. Sadovsky, R. Schicker, A. Shabanov, F. Sikler, A. Villatoro Tello, T. Tuunanen, J. Welti. Last but not least, I would like to thank my family and friends for their support.

I acknowledge the funding by Helsinki Institute of Physics and University of Helsinki. Now it is time for some new challenges in the HEP Group at the Blackett Laboratory of Imperial College London.

London, January 2020

Work of the Author

All computational machinery and code of this work are available as Open Source under the MIT license at github.com/mieskolainen for maximal scientific impact, scholarship and reproducability. *Every* single figure or number in this thesis is produced by this code together with external libraries.

Diffraction phenomenology and computational theory

1. GRANIITTI: A new Monte Carlo event generator and algorithmic engine written in fully multithreaded C++17 with 35k lines of code. Currently the most advanced event generator for the low mass central exclusive diffraction at the LHC and beyond.
   Motivation: Explicit control of a Monte Carlo event generator capable of simulating spin dependent scattering processes in the low mass central diffraction with forward proton excitation kinematics.

2. COMBINATORIAL SUPERSTATISTICS: A new statistical incidence algebra approach, Möbius inversion and abstract construction designed as a higher dimensional definition of diffractive event topologies and substructures.
   Motivation: Abstract theory beyond standard tools and a self-consistent measurement framework beyond the traditional large rapidity gap counting.

Advanced algorithms

1. DEEPEFFICIENCY: The first Deep Learning based detector efficiency inversion algorithm for maximally unbiased fiducial measurements in higher dimensions.
   Motivation: Maximally Monte Carlo event generator independent detector efficiency corrections of single or multidimensional observables.

2. KISU: A new Shannon entropy and the Fast Fourier Transform based algorithm for non-linear inversion of stochastic autoconvolution integral equations. The first application with public ALICE data.
   Motivation: Algorithmic way to invert statistically multiparton interactions, multipomeron interactions or pileup.

3. DREM-PID: A new two-step probabilistic particle identification algorithm based on Expectation Maximization (EM) iteration.
   Motivation: Mathematically optimal final state identification.

4. F^*-PROJECTION: Overcomplete MC basis projection technique relying on the combinatorial superstatistics framework.
   Motivation: Multidimensional data projections with limited detector information.

5. S-HARMONICS: A new formulation of the classic Laplace spherical harmonics decay distribution decomposition with novel detail in defining different phase spaces: fiducial versus flat non-fiducial and their invertability and harmonic mixing properties in different Lorentz rest frames.
   Motivation: Mathematically rigorous angular distribution and spin polarization measurements.

ALICE and *beyond*

1. *The first* measurement of unfolded and fiducial *combinatorial cross sections* at the LHC, done with the ALICE proton-proton data at the center-of-mass energy $\sqrt{s} = 13$ TeV. The mathematical construction and analysis technology built up in this thesis.
   Motivation: Maximally model independent, multidimensional fiducial measurement of diffraction and soft QCD.

2. *The first* fiducial multidimensional maximum likelihood extraction of single, double and non-diffractive cross sections. A factorized definition of fiducial cross sections and then extrapolation to total inclusive cross sections.
   Motivation: Mathematically nearly optimal extraction of the model based cross sections, separate fiducial and extrapolated total cross sections.

3. *The first* multidimensional maximum likelihood extraction of the effective pomeron intercept and the rapidity gap distributions with incomplete forward detector information.
   Motivation: High precision diffraction phenomenology with incomplete data.

4. Resonance spectrum measurements of semi-exclusive diffraction at the LHC with the ALICE proton-proton data at the center-of-mass energy $\sqrt{s} = 7$ TeV. We introduce also a new analysis and generic data preservation strategy called *F2X*.
   Motivation: Where are the glueballs and what happens with the low mass forward proton dissociation?

List of publications

1. M. Mieskolainen
   GRANIITTI: A Monte Carlo Event Generator for High Energy Diffraction
   arXiv:1910.06300 [hep-ph]

2. M. Mieskolainen
   Combinatorial Superstatistics for Soft QCD
   arXiv:1910.06279 [hep-ph]

3. M. Mieskolainen
   On the Inversion of High Energy Proton
   arXiv:1905.12585 [hep-ph]

4. M. Mieskolainen
   DeepEfficiency – optimal efficiency inversion in higher dimensions at the LHC
   arXiv:1809.06101 [physics.data-an]

5. M. Mieskolainen
   Algorithmics of Diffraction
   Acta Physica Polonica B Proceedings of Diffraction and Low-x 2018
   arXiv:1811.01730 [hep-ph]

6. M. Mieskolainen, R. Orava
   Observables of QCD Diffraction
   AIP Proceedings of Diffraction 2016
   arXiv:1612.00980 [hep-ph]

7. The MoEDAL collaboration
   Magnetic Monopole Search with the Full MoEDAL Trapping Detector in 13 TeV *pp* Collisions Interpreted in Photon-Fusion and Drell-Yan Production
   Physical Review Letters 123, 021802 (2019)
   arXiv:1903.08491 [hep-ex]

8. K. Akiba et al.
   LHC Forward Physics
   Journal of Physics G: Nuclear and Particle Physics 43 110201 (2016)
   arXiv:1611.05079 [hep-ph]

Contributions in papers with multiple authors

* In 6: I wrote the manuscript, came up with the incidence algebras and did the analytical and numerical work.

* In 7: I contributed by re-implementing independently the photon fusion based generator level simulation of spin-1/2 monopole pairs and thus re-checked the numbers in the paper before the publication. This implementation with velocity β-dependent and pure Dirac couplings, driven by collinear EPA, k_t-EPA and inclusive luxQED-pdf photon fluxes, is publicly available in GRANIITTI event generator.

* In 8: I contributed in the AD detector project and in the ALICE diffraction analysis group in which I participated in detector wrappings, in a beam test, low-level signal analysis, high-level physics analysis, a double rapidity gap trigger design planning and verified detector simulations.

Contents

1 Introduction

Algorithms and high energy physics are dual topics. The first modern computer architecture was introduced by John von Neumann in 1945 [1], the same man behind axiomatic quantum mechanics. The Manhattan project resulted in the first Monte Carlo sampling methods by Stanislav Ulam and von Neumann. The best known physics algorithms are due to Richard Feynman [2] – the path integral description and the diagrammatic representation of the perturbation series, also the most visual pictures of fundamental interactions. The first large scale computer algebra software SCHOONSCHIP [3] was written by Martinus J.G. Veltman in 1964 in studies towards what later resulted in the renormalizability proof of Yang-Mills [4] theories by Gerard 't Hooft and Veltman. The discretized integral transform by Paul Hough in 1962 [5] was the first computer vision AI-algorithm utilized first in the bubble chamber track fitting, nowadays used by self-driving cars to keep the car between the highway lines or to geometrically align your favorite urban Instagram pictures.

Stephen Wolfram, also from high energy physics, industrialized computer algebra with Mathematica [6] in the 1980's and the World Wide Web was invented at CERN by Tim Berners Lee [7] to organize the experimental information chaos. LHC experiments produce more data faster than any other scientific experiment so far. Also, by physics standards, the most heterogeneous data ever. The ROOT technology [8] developed at CERN during 1990's was during its launch the fastest fully generic object data to disk serializer in the world, perhaps still is. Software like MadGraph have automated computations that have seemed impossible. Nowadays much-hyped quantum computers and their true supremacy are seemingly the distant future of computation, discussed first by Feynman in 1982 [9]. However, the future may rely instead on synthetic biology. Artificial neural networks are currently the mainstream target of large scale computer algebra, more precisely, the network optimization relies on the so-called automatic differentiation techniques, which we have also utilized in this thesis. What is not always known that the crucial reverse-mode automatic differentiation of the current AI industry, later re-invented as the backpropagation, was first invented by a Finnish mathematician Seppo Linnainmaa from Helsinki.

This was in 1971 during his MSc thesis, published later in [10]. Of course, also pure mathematics gets its part from high energy physics, in terms of string theory and scattering amplitude techniques, such as generalized polylogarithms. To summarize, we have listed some down to earth examples to motivate the studies of high energy interactions.

According to the de Broglie particle-wave duality, all matter must exhibit wavelike properties with the wavelength $\lambda = h/p$ inversely proportional to the momentum p, with the bounding constant of fundamental resolution being the Planck constant h. This is natural given the uncertainty principle behind non-commuting observables or the Fourier transform, on the other hand. The wavelike quantum properties of matter span all scales of physics. Experimentally the non-classical behavior has been recently observed even with a chain of 15 amino acid biomolecules [11]. However, we do not know what happens with quantum gravity or is the space-time discretized, for example.

The first documented diffraction observations are from 1665 by Francesco Maria Grimaldi [12]. Perhaps the most famous quantum mechanics experiment is the electron diffraction through the double slit, which demonstrates the probabilistic Born rule and wavelike properties of elementary particles in a highly controlled way. The double helix structure of DNA was discovered by using X-ray diffraction crystallography. Everything of diffraction in terms of quantum electrodynamics (QED) is well understood. However, the Born rule, for example, cannot be currently derived but is an axiom, which fits the data. Many algebraic arguments have been developed to support it. It works remarkably well and there are no known deviations of it. Experiments to test deviations of Born rule are often based on multi-slit diffraction. According to the path integral picture, particles are free to go several loops around the slits in space-time. However, the weight of these highly non-classical paths is usually vanishing in the full probability amplitude.

The topic of this work – hadronic high energy diffraction, is far from well understood. This is due to the non-perturbative strong nature of quantum chromodynamics (QCD), N-body parton problems and the peculiar nature of confinement. That is, there are no currently known powerful ab initio methods to simulate soft QCD diffraction, such as lattice methods or small parameter expansions. Phenomenological Monte Carlo modeling is the only way to get proper observables simulated for the comparison with data. However, there are some deep principles behind this modeling, in the context of Regge theory, relativistic kinematics and spin algebra, for example. It is not just parametrizations of data. In general, one needs to remember that we cannot currently even calculate the proton structure function or parton density input at the starting scale for any kind of events, not only diffractive. However,

once the input is fitted from data, the integro-differential evolution schemes such as DGLAP work quite well together with hard matrix elements and parton showers, and reliable predictions can be made for the LHC and elsewhere to hunt for new particles. This is based on a factorization between the soft and hard scales, which is not always exact, but extremely useful nevertheless.

1.1 Strong and even stronger interactions

The primary tool to probe strong interactions at ever-increasing energies is a high energy collider and a general purpose detector built around the interaction point. More and more powerful microscopes are being built, in essence. The initial state may be either leptons, hadrons or heavy ions. The choice of the particular initial state is dictated by the physics goals. High energy collider experiments have a long history at CERN, Brookhaven, SLAC, Berkeley and Protvino. The colliders highly relevant regarding this work are the ISR [13] at CERN, the S$p\bar{p}$S [14] at CERN, the LEP [15] at CERN, the HERA [16] at DESY, the Tevatron [17] at Fermilab and nowadays the RHIC [18] at Brookhaven and the LHC at CERN [19, 20]. The future is unknown, but large electron-ion colliders and massive proton-proton colliders may be expected.

The non-abelian $SU(3)$ gauge theory of strong interactions [21–24], quantum chromodynamics, is very complicated. Ideally, we would like to understand it at a similar level of detail as the quantum electrodynamics is understood. Yet understanding QED in detail does not automatically mean we understand collective N-body phenomena, such as chemistry, but only in principle. In the numerous corners of strong interactions, there are many new techniques, but the soft Regge domain has basically always defeated any other methods than the Regge-like power law scaling asymptotics or calculus based on it. There simply seems not to exist any truly powerful small parameter to expand against, other than already found decades ago. The original Reggeon field theory by Gribov was left unfinished by the master. Naturally, many papers have been written to complete it in several plausible directions. However, one should not limit oneself to this massive obstacle posed by field theories – measuring, probing and modeling QCD diffraction is possible in various other ways.

Our number one strategy in this work is to design rigorous mathematical algorithms and then using them, implement novel higher dimensional measurements. Of course, along the way we want to see in explicit detail, what is possible in terms of models, how they compare with the LHC data and introduce an advanced simulation framework, Graniitti, with special emphasis on spin. For the mathematical formulation of inclusive diffraction observables, our approach is to use incidence algebras. These algebras were formulated by the father of modern combinatorics, mathematician Gian-Carlo Rota at MIT in the 1960's. That is, we use a simple combinatorial counting of final states as our starting point. Simple is good because already the experimental issues are very complex. Basically, what we formulate is a new definition of diffraction. Funnily enough, this picture is visually closer to the classic diffraction

Soft QCD	Low-$p_t \sim$ Asymptotic energy dependence, elastic scattering, inelastic diffraction: single, double and central diffraction, multiple large rapidity gaps, spin-parity selection rules in central diffraction, absorption and screening, multiplicity, transverse momentum: from exponential to power laws, soft multiparton (multipomeron) interaction observables, transition to hard scattering, long range rapidity correlations (ridge structure) and high multiplicity without jets, input for fragmentation (hadronization) models and functions
Hard QCD	High-$p_t \sim$ Deep(ly) inelastic scattering, differential jet and system kinematics, event shapes, jet composition, substructures, multijets, radiation patterns and color flow effects, quark/gluon separation, IR/CL-safe jet algorithms (hard, weighted), parton density $f(x, Q^2)$ fitting, hard (HERA) diffraction = hard system + rapidity gap, hard multiparton interactions, search for new massive BSM-resonances, running coupling α_s measurements
Cosmic Rays	Extended Air Showers (EAS), very forward 'fragmentation region' measurements of energy and particle composition at colliders, Monte Carlo tuning
Heavy Ions	Probing the unknown QCD phase diagram at different densities and temperatures, chiral symmetry recovery, search and understanding for the solid observables of Quark-Gluon Plasma (QGP): photon and lepton rates, strangeness, quarkonia, jet quenching, plasma screening effects, spherical and elliptic flow, fluctuations, Bose-Einstein correlations, phase transition temperature fits, search for GLASMA and other hypothetical (amorphic) states of matter
Spectroscopy	Light mesons, baryons, glueballs and their mixing, multiquark states
Quarkonia	$J/\Psi(c\bar{c}), \Upsilon(b\bar{b}), \ldots$ and their spin-excited states
Spin Physics	Quarkonia polarization, polarization in photoproduction Gamma-Pomeron ($\gamma - gg$) processes, proton pdf 'spin crisis'

Table 1.1: Strong interactions topics classified by experimental observables.

Nuclear	Yukawa model (30's), Chiral perturbation theory (60's), Effective theories
Regge domain	Regge theory (60-70's), Hard domain (Lipatov et al. since late 70's), Durham QCD (KMR) type models (00's), Stochastic calculus (70's)
Low-x	Saturation (80's), Color Glass Condensate (90's), Classic Yang-Mills
Integro-Differential	DGLAP, BFKL, BK ... (70's - 90's)
Parton densities	Bjorken scaling, Feynman-Gribov parton model (late 60's), QCD integrated pdfs, unintegrated (generalized) pdfs (80-90's)
Collider QCD, jets	Fixed order pQCD (late 70's), multileg/multiloop 'NLO revolution' (late 00's), analytic resummation, parton showers, Monte Carlo event generators (late 70's), effective collinear field theories (00's)
High Density	Cold and hot nuclear matter, the equation of state (EOS), Neutron stars, thermal-pQCD + Lattice QCD (70's)
Hadron spectroscopy	Regge trajectories (60's), Gell-Mann/Zweig quark/ace model (60's), Lattice QCD, Holography (90's), Supersymmetric meson-baryon spectrum (00's)
Quarkonia	Non-Relativistic NR-QCD (80's)
Hadronization	Lund strings (late 70's), Pre-confinement (Veneziano), Webber clusterization (80's)
Vacuum properties	Instanton calculus and 't Hooft, Wilson lattice QFT, lattice QCD (late 70's)
Hydrodynamics	Lattice, transport coefficients (80's)
Scattering amplitudes	S-matrix unitarity and analyticity (60's), Spinor-Helicity / Parke-Taylor (80's), Generalized unitarity (90's), Color factorized amplitudes and string theory methods (00's), geometric 'Amplituhedron' methods (10's)

Table 1.2: Strong interactions topics classified in a theory driven way, with the approximate time of origin indicated.

pattern experiments than the de facto 'large rapidity gap' counting. What we also see is that it should be one of the best ways to probe the algebraic properties of the Abramovski-Gribov-Kancheli (AGK) scattering amplitude cutting rules [25]. That is, how does one see those rules from data? The interference structure contained in this calculus should be reflected in the multiplicity densities per rapidity interval. Those rules are, after all, of highly combinatorial nature.

To this end, we may mention that increasingly many collider measurements are collected under the automated Rivet [26] platform and their comparison with numerous Monte Carlo models is algorithmized under mcplots.cern.ch [27]. A technical requirement for this to work is that the measurement is a strictly fiducial one. Our novel measurements are also fiducial measurements and thus directly comparable with event generators using only the final state information. We see that highly automated and fiducial approaches should be the case for all fields of physics and science in general, not just limited to collider physics. We shall illustrate for the reader the topics of strong interactions in Tables 1.1 and 1.2. Obviously, we ignore the (effective) strong interactions in condensed matter and elsewhere and stay only in the high energy physics context.

1.2 Fundamental open problems

To expand the mind of the reader regarding where are our topic fits in the large spectrum of modern physics, we shall first list the following to span the space of outstanding problems and questions, in no specific strong order. Our topic belongs to the first one.

- ◇ **Non-perturbative strong interactions and scattering amplitudes**
- ◇ Detectors operating at the single quantum limit at different energy scales
- ◇ Early universe, big bang, inflation, monopoles and topological defects
- ◇ Origin of mass, hierarchy, flavor, matter versus antimatter and details of (the) Higgs boson
- ◇ Unification of dynamics, superstrings, extra dimensions and holography
- ◇ The cosmological constant, dark energy and vacuum(s)
- ◇ Dark matter or misunderstood gravity at large scales
- ◇ Quantum gravity, black holes and information, wormholes and entanglement
- ◇ Stochastic gravitational waves in the (early) universe
- ◇ Quantum computers; unitary port logic driven versus 'adiabatic' realizations
- ◇ Algorithms and the synthesis of biology, the arrow of time and entropy
- ◇ Artificial intelligence and physics of neuroscience; classical versus quantum
- ◇ Mathematical physics: number theory and physics, generalized particle statistics, topics of string theory
- ◇ Highly geometric theory of quantum mechanics and space-time; twistors, amplituhedrons, emergent unitarity, explanations for the origin of gauge symmetries
- ◇ High temperature superconductivity, exotic forms of condensed matter and quantum chemistry

1.3 Outline

The structure of this thesis goes as follows. In Chapter 2 we go through the classic picture of diffraction in terms of waves and paths and illustrate these visually brilliant topics through simulations. In Chapter 3 we described the main measurement and multidimensional diffractive cross section extractions of this thesis. In Chapter 4 we discuss new algorithms for semi-exclusive diffraction and glueball hunting and go through case studies with data. In Chapter 5 we have a computational tour of high energy diffraction, where we describe our new Graniitti Monte Carlo event generator and advanced spin analysis tools. In Chapter 6 we describe in detail the mathematics of our combinatorial measurement framework and the related inverse problems. In Chapter 7 we study the 'inversion of proton', by first developing a new inverse algorithm for stochastic autoconvolution integral equations and then apply it to data. In Chapter 8 we describe the first Deep Learning based high dimensional detector efficiency inversion algorithm, DeepEfficiency. Finally in Chapter 9 we end with overall conclusions.

2 Waves and Paths

We shall remind ourselves of the elegant formulations of classic diffraction. There are no radical new results in this chapter, only a compact summary of the essentials.

Huygens (1678)–Fresnel (1818) principle Every space-time point of a propagating wavefront is a source of secondary spherical wavefronts (recursion).

Babinet's principle (∼1800) A geometric aperture with a hole gives the same diffraction pattern in the *far field* as the complement aperture (geometry).

2.1 Wave and Helmholtz equation

From the Maxwell equations [28], we can derive the vector valued wave equation [29]

$$\nabla \times (\nabla \times \mathbf{E}(\mathbf{x}, t)) + \frac{1}{v^2} \frac{\partial^2}{\partial t^2} \mathbf{E}(\mathbf{x}, t) = 0, \tag{2.1}$$

which is a second order partial differential equation (PDE) in space and time with the speed of propagation v. Now, the corresponding scalar wave equation reads

$$\left(\nabla^2 - \frac{1}{v^2} \frac{\partial^2}{\partial t^2} \right) U(\mathbf{x}, t) = 0, \tag{2.2}$$

where the scalar function $U(\mathbf{x}, t)$ can be taken one of the spatial components of $\mathbf{E}(\mathbf{x}, t)$. By using the scalar equation, we loose all the polarization (vector) dependent phenomena, which is just fine in the case of acoustic fields, for example.

Then substitution of a single monochromatic wave $U(\mathbf{x}, t) = u(\mathbf{x}) e^{-i\omega t}$ at angular frequency ω gives us the linear Helmholtz equation with no time dependence [30]

$$\left(\nabla^2 + k^2 \right) u(\mathbf{x}) = 0, \tag{2.3}$$

where the wavenumber is $k^2 = \omega^2 / v^2$, $k = 2\pi/\lambda$. Usually, one uses Dirichlet and von Neumann boundary conditions. The former sets u on the boundary and the latter defines $\partial u / \partial \mathbf{n}$ on the boundary. A practical way to solve classic diffraction or field configurations in arbitrary geometries and materials is to simulate the scalar fields or Maxwell field equations numerically. An often used formulation in electrodynamics is the Finite Difference Time Domain (FDTD) method invented by Yee in 1966 [31]. However, we shall go through certain analytic classic scalar diffraction theory results. To this end, we may dream about the future where high energy QCD quantized non-linear field equations can be solved on a computer like Maxwell equations.

2.2 Helmholtz-Kirchoff integral theorem

Coordinates: Let our aperture plane and the coordinate system xy-plane coincide with a positive z-axis taken towards the detector. Let $\mathbf{x}$ be the position vector where we evaluate the diffraction integrals at the aperture and let $\mathbf{y}$ be the position vector in the outgoing space at the virtual detector.

We use the Green's [32] equation for a point source

$$\left(\nabla^2 + k^2\right) G(\mathbf{x}, \mathbf{y}) = -4\pi\delta(\mathbf{x} - \mathbf{y}) \tag{2.4}$$

with $G(\mathbf{x}, \mathbf{y})$ being the Green function kernel

$$G(\mathbf{x}, \mathbf{y}) = \frac{e^{ik|\mathbf{x}-\mathbf{y}|}}{|\mathbf{x} - \mathbf{y}|}. \tag{2.5}$$

In this case we have translation invariance $G(\mathbf{x} - \mathbf{y}) \equiv G(\mathbf{x}, \mathbf{y})$ by homogeneous medium, such as free space or dielectric. Thus, the solution can be written as a convolution

$$u(\mathbf{y}) = \frac{1}{4\pi} \int_V d\mathbf{x}\, G(\mathbf{x} - \mathbf{y}) s(\mathbf{x}), \text{ if } \mathbf{y} \in V \text{ and } u(\mathbf{y}) = 0 \text{ for } \mathbf{y} \text{ outside } V, \tag{2.6}$$

for the equation $\left(\nabla^2 + k^2\right) u(\mathbf{x}) = -s(\mathbf{x})$ within volume V.

The Helmholtz-Kirchoff (HK) integral theorem [33] states that we can express the scalar field at a point $\mathbf{y}$ as a function of the field values at the volume boundary ∂V

$$\text{HK theorem: } u(\mathbf{y}) = \frac{1}{4\pi} \int_{\partial V} dS(\mathbf{x}) \left(u(\mathbf{x}) \frac{\partial G}{\partial \mathbf{n}}|_{\mathbf{x}-\mathbf{y}} - G(\mathbf{x} - \mathbf{y}) \frac{\partial u}{\partial \mathbf{n}}|_{\mathbf{x}} \right), \tag{2.7}$$

where $\mathbf{n}$ is a unit normal vector oriented to inside the volume V with normal derivative $\partial u/\partial \mathbf{n} \equiv \nabla u(\mathbf{x}) \cdot \mathbf{n}$ and $S(\mathbf{x})$ is a surface element. In addition, we need boundary conditions [29]

$$\mathcal{A} : \text{Kirchoff aperture} : u(\mathbf{x}) = u_I(\mathbf{x}) \ \wedge \ \frac{\partial u}{\partial \mathbf{n}} = \frac{\partial u_I}{\partial \mathbf{n}} \tag{2.8}$$

$$\mathcal{S} : \text{Kirchoff screen} : u(\mathbf{x}) = 0 \ \wedge \ \frac{\partial u}{\partial \mathbf{n}} = 0 \tag{2.9}$$

$$\mathcal{R} : \text{Sommerfeld radiation} : \lim_{|\mathbf{x}| \to \infty} |\mathbf{x}| \left(\frac{\partial u}{\partial \mathbf{n}} - iku(\mathbf{x}) \right) = 0, \tag{2.10}$$

where we denote the aperture (hole + obstacle) with $\mathcal{A}$, the opaque screen or detector with $\mathcal{S}$ and the outgoing far field radiation field half-sphere boundary with $\mathcal{R}$. The incoming field is denoted with $u_I(\mathbf{x})$.

2.3 Fresnel-Kirchoff and Rayleigh-Sommerfeld formulations

Now we use a spherical point source

$$u(\mathbf{x}) = A\frac{e^{ik|\mathbf{x}-\mathbf{x}_s|}}{|\mathbf{x}-\mathbf{x}_s|} \equiv A\frac{e^{iks}}{s} \tag{2.11}$$

with the source position $\mathbf{x}_s$ and the amplitude A. For the Green function, we get the gradient as

$$\frac{\partial G}{\partial \mathbf{n}}|_{(\mathbf{x}-\mathbf{y})} = -\left(ik - \frac{1}{|\mathbf{x}-\mathbf{y}|}\right) G(\mathbf{x}-\mathbf{y})\cos\delta(\mathbf{x},\mathbf{y}) \tag{2.12}$$

$$\simeq -ikG(\mathbf{x}-\mathbf{y})\cos\delta(\mathbf{x},\mathbf{y}), \tag{2.13}$$

where the approximation holds when $|\mathbf{x}-\mathbf{y}| \gg \lambda$. Above the so-called inclination factor is

$$\cos\delta(\mathbf{x},\mathbf{y}) = \mathbf{n}_\mathbf{x} \cdot \left(\frac{\mathbf{x}-\mathbf{y}}{|\mathbf{x}-\mathbf{y}|}\right), \tag{2.14}$$

where δ is the angle between the vector $\mathbf{x}-\mathbf{y}$ and the normal vector $\mathbf{n}_x$ (z-axis). Then using the HK theorem, we get the Fresnel-Kirchoff diffraction equation [34] as

$$\text{FK equation:}\quad u(\mathbf{y}) = -\frac{1}{4\pi}\int_\mathcal{A} dS(\mathbf{x})\left(iku(\mathbf{x})\cos\delta(\mathbf{x},\mathbf{y}) + \frac{\partial u}{\partial \mathbf{n}}|_\mathbf{x}\right) G(\mathbf{x}-\mathbf{y}). \tag{2.15}$$

However, this equation is ill-posed by inconsistent boundary conditions, presumably first noted by Sommerfeld. We would like to get rid of the normal derivative $\partial u/\partial \mathbf{n}$ term at the aperture.

Getting rid of the normal term is done by imposing the Sommerfeld radiation condition of Eq. 2.10 to the FK equation and evaluating Eq. 2.12 at the point $\mathbf{x}$ instead of $\mathbf{x}-\mathbf{y}$, which results in a factor of 2 difference. These modifications yield the Rayleigh-Sommerfeld [35] diffraction equation

$$\text{RS equation:}\quad u(\mathbf{y}) = -i\frac{k}{2\pi}\int_\mathcal{A} dS(\mathbf{x})\, u(\mathbf{x})\cos\delta(\mathbf{x},\mathbf{y})G(\mathbf{x}-\mathbf{y}). \tag{2.16}$$

Note that sometimes in the literature, this is called the Fresnel-Kirchoff equation. This integral is a superposition of source waves (Huygen's principle) with the phase shift at the aperture by $\pi/2$ dictated by $-i$ factor. We see that the boundary conditions restricted the Helmholtz-Kirchoff integral to be non-zero only on the aperture

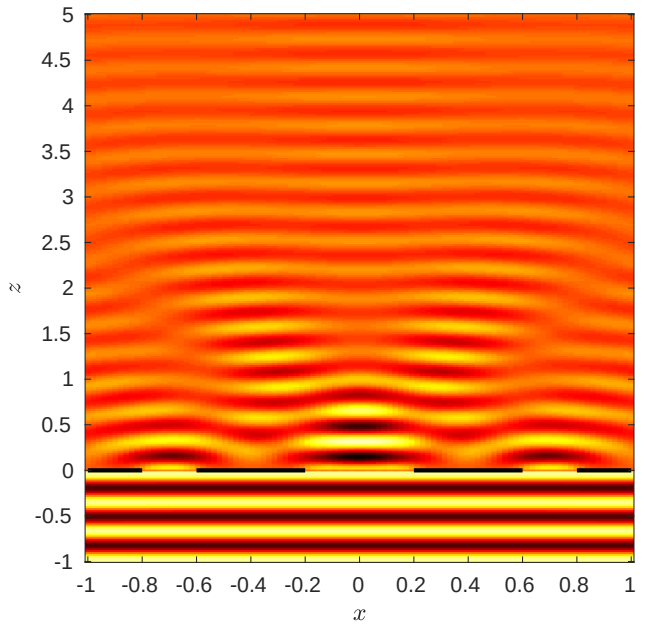

Figure 2.1: A numerical Rayleigh-Sommerfeld integral based simulation of the field values $u(\mathbf{y})$ with three square holes in $\mathcal{A}$, the source being isotropic at $z \to -\infty$.

boundary $\mathcal{A}$. That is, the field value at $\mathbf{y}$ depends only on that. One could call this the first *holographic principle*, later made explicit by Gabor by the invention of optical holography in 1947 and then extended to extra space-time dimensions by 't Hooft, Susskind, Maldacena and others.

2.4 Fresnel and Fraunhofer approximations

By translation invariance and some approximations, we can re-write the RS diffraction equation as a convolution integral

$$u(\mathbf{y}) = \int_{\mathcal{A}} dS(\mathbf{x})\, u(\mathbf{x}) h(\mathbf{x}-\mathbf{y}), \tag{2.17}$$

using the kernel function $h(\mathbf{x})$. Now different classic analytic approximation schemes can be obtained by expanding the kernel in various ways and truncating, for example in the so-called Fresnel approximation

$$\text{Fresnel regime: } \frac{b^2}{z\lambda} \sim 1, \tag{2.18}$$

where b^2 is approximately the aperture dimension squared, $z \simeq |\mathbf{x}-\mathbf{y}|$ is the distance from the aperture to the detector and λ is the wavelength of the incoming radiation. Actually, by reciprocity of the FK- or RS-equation, the condition assumes also that z can be replaced by $s = |\mathbf{x}-\mathbf{x}_s|$, which is the distance from the source to the aperture. The corresponding approximated Green's function kernel is

$$h(\mathbf{x}-\mathbf{y}) = \frac{1}{i\lambda z} e^{ikz} e^{\frac{ik}{2z}|\mathbf{x}-\mathbf{y}|^2} \simeq -i\frac{k}{2\pi}\cos\delta(\mathbf{x},\mathbf{y}) G(\mathbf{x}-\mathbf{y}) \tag{2.19}$$

obtained by Taylor expanding

$$|\mathbf{x}-\mathbf{y}| \simeq z\left(1+\frac{1}{2}\left[\frac{|\mathbf{x}-\mathbf{y}|}{z}\right]^2\right) \tag{2.20}$$

inside the exponential function to the first non-trivial order and outside exponential simply with $|\mathbf{x}-\mathbf{y}| \simeq z$ and substituting these in the Green's function of Eq. 2.5. The inclination factor was approximated with 1 in the forward (paraxial) limit $\delta \to 0$, such that the vector $\mathbf{y}$ has much larger z component than transverse ones. The Fresnel diffraction integral equation is then

$$\begin{aligned}\text{Fresnel equation: } u(\mathbf{y}) &= \frac{1}{i\lambda z} e^{ikz} \int_{\mathcal{A}} dS(\mathbf{x})\, u(\mathbf{x}) e^{\frac{ik}{2z}|\mathbf{x}-\mathbf{y}|^2} \\ &= \frac{1}{i\lambda z} e^{ikz} e^{\frac{ik}{2z}|\mathbf{y}|^2} \int_{\mathcal{A}} dS(\mathbf{x})\, u(\mathbf{x}) e^{\frac{ik}{2z}|\mathbf{x}|^2} e^{-\frac{ik}{z}\mathbf{x}\cdot\mathbf{y}}.\end{aligned} \tag{2.21}$$

If the source and the detector are in the *far field* from the aperture or we model the outgoing field from a lens which is focusive (positive), we can model the diffraction

of the waves as plane waves and use the Fraunhofer approximation which results in linear dependence on the integration variable on the aperture, that is, a 2D-Fourier transform. The key point here is that the phase of the field is the same at each point at the aperture, due to incoming plane waves. The approximation condition can be written as

$$\text{Fraunhofer limit:}\quad \frac{b^2}{z\lambda} \ll 1, \tag{2.22}$$

where z can again be replaced with s. The Fraunhofer diffraction integral equation is a simplified version of the Fresnel equation

$$\text{Fraunhofer equation:}\quad u(\mathbf{y}) = \frac{1}{i\lambda z} e^{ikz} e^{\frac{ik}{2z}|\mathbf{y}|^2} \int_{\mathcal{A}} dS(\mathbf{x})\, u(\mathbf{x}) e^{-\frac{ik}{z}\mathbf{x}\cdot\mathbf{y}}, \tag{2.23}$$

obtained by neglecting the quadratic terms from the expansion, to obtain linear dependence on the integration variables. Classic analytic solutions can be obtained by Fourier transform for rectangular ($\rightarrow \text{sinc}^2$), circular ($\rightarrow$ Airy) and Gaussian ($\rightarrow$ Gaussian) density slits.

2.5 Feynman path integral

The Feynman path integral based [2] complex transition amplitude or propagator for a particle, to start from (x_i, t_i) and end up in (x_f, t_f), in a non-relativistic formulation is

$$K(x_f, t_f|x_i, t_i) = \langle x_f, t_f|x_i, t_i\rangle = \int_{\text{all paths}} D[x(t)]e^{iS[x(t)]/\hbar}, \tag{2.24}$$

where all different paths of the integral represent different quantum *phases*. The transition probability is obtained as the amplitude squared, as usual, by the Born rule. The Born rule cannot be derived, currently. In the transition amplitude, the action functional is a time integral over the Lagrangian, which encapsulates the dynamics of our physics

$$S[x(t)] = \int_{t_i}^{t_f} L[x(t), \dot{x}(t), t]\, dt \tag{2.25}$$

with a classical Lagrangian, here

$$L = \frac{1}{2}\dot{x}(t)^2 - V[x(t), t], \tag{2.26}$$

where V is an external potential term and the dotted variable is the time derivative. The Feynman path measure $D[x(t)]$ needs to be understood as the following discretization

$$D[x(t)] \equiv \lim_{n\to\infty} \frac{1}{(2\pi i\hbar\Delta/m)^{d/2}} \int_{\mathbb{R}^d} \prod_{k=1}^{n-1} \frac{dx_k}{(2\pi i\hbar\Delta/m)^{d/2}}, \tag{2.27}$$

with the phase oscillating exponentiated action discretized as

$$\exp(iS[x(t)]/\hbar) \to \exp\left(\frac{i}{\hbar}\sum_{z=1}^{n}\left[\frac{1}{2}m\left(\frac{x_z - x_{z-1}}{\Delta}\right)^2 - V(x_z)\right]\right), \tag{2.28}$$

where d denotes the number of spatial dimensions, $x_0 \equiv x_i$, $x_n \equiv x_f$ and $\Delta = (t_f - t_i)/n$. For a mathematician, this path integral measure causes some headache, especially in the case of quantum field theories. The stochastic path integral with the Wiener measure, on the other hand, is well defined. This is probably just lack of suitable mathematics. Physically, this picture is both intuitive and elegant. It also provides the path to lattice quantum field theories.

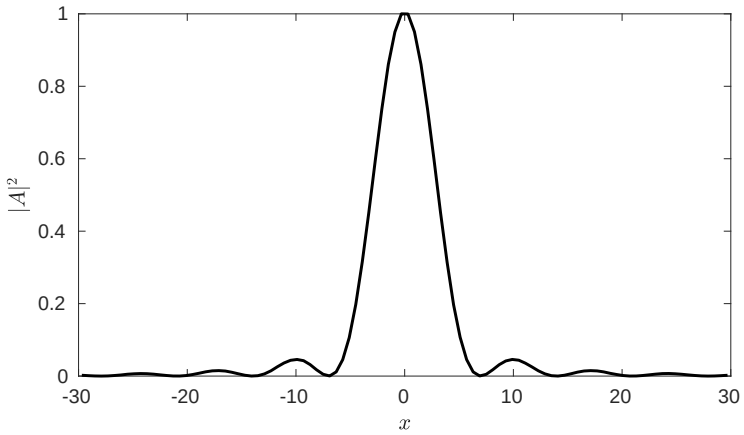

Figure 2.2: A toy Monte Carlo complex path integral simulation of diffraction.

Now a simple single or double slit diffraction can be trivially calculated in 1+1 dimensions for a free particle without any potential term. Instead of an analytical calculation, by curiosity, we did a numerical Monte Carlo simulation, where instead of doing any Wick rotation to imaginary time as is often plausible to do to avoid the highly oscillating exponential term, we brute force evaluated directly the complex path integral in real time by sampling and accumulating discretized paths without any importance sampling. The resulting diffraction pattern is illustrated in Figure 2.2. For proper simulations, one needs to remember fine enough spatial sampling in the propagation and at the detector by the Shannon-Nyquist-Whittaker-Kotelnikov sampling theorem, in order not to produce aliasing. Extreme discretization instability was observed with complex path sampling, a major problem in different quantum simulation scenarios known in general as the *sign problem*. It is an NP-hard problem with no known generic solutions [36]. Basically, this is currently *one* of the fundamental limitations to any ab initio simulations of high energy diffraction and suggests the need for quantum computers, not feasible yet.

On the other hand, deep learning based approximation techniques are already capable of producing extremely high dimensional generative sampling of photorealistic high resolution images [37]. This could be an interesting target for the future research regarding the lattice simulations. This is because image matrices can be understood analogous to extremely complicated lattice field configurations.

3 ALICE and *combinatorial measurement*

We describe the proton-proton combinatorial fiducial cross section measurement at the center-of-mass energy of $\sqrt{s} = 13$ TeV in the ALICE experiment, implemented during the thesis. The full analysis code and grid computing code, including all experimental treatments, is available online on github. It is directly compatible with the ALICE experiment, but all fundamental algorithms and methods are experiment independent, by design. The measurement is the first of its kind, no similar multidimensional unfolded measurement has been attempted before. However, we shall point out that the TOTEM double diffractive measurement [38] is technically a subset of this measurement. Also already the measurements by UA5 [39] were towards this direction, however, philosophically different. Here our main goal is *not* actually to extract diffractive cross sections, but go beyond and implement a fiducial measurement of the combinatorial subspaces or measure the Grasmannian, a mathematician would say. From another perspective, these cross sections are in a sense *coded diffraction aperture* measurements – a high energy analog to the classic case of controlled coded aperture diffraction, which can be used for the phase field recovery with modern algorithms. Here we use them to recover high energy observables and model parameters.

A detailed description of the developed methodology is given in Chapter 6. In the analysis, all basic low level detector distributions were compared with the GEANT simulations driven by Monte Carlo event generators, low level detector signal quality cuts were part of the basic procedures and explicit cut flows were inspected. We shall concentrate here on the physics results but also go through all the basic steps. This and our code should inspire also others to implement a radically new type of measurements.

3.1 Experimental setup

The detector setup consisted of the AD, VZERO and SPD subdetectors, with maximal pseudorapidity coverage. The minimum bias trigger required was algebraic 'global OR', where each of the subdetectors had their independent signal decision criteria. The ALICE experiment uses a naming convention of C- and A-sides, where the C-side is on negative rapidities and the A-side on positive. For the generic detector performance of the ALICE experiment, see [40].

The VZERO detector [41] is a forward scintillating plastic counter made with eight cells in four radial rings giving 32 channels per rapidity side. Scintillator detectors are based on a physical mechanism where incoming charged particles induce molecular excitations in the plastic, which is de-excited by the emission of visible wavelength photons. Hamamatsu photomultipliers (PMT) are coupled directly to the scintillating plastics on the A-side and on the C-side through optic fibers. The digitized output gives signal hit time and accumulated charge information. The geometric acceptances over pseudorapidity are $\eta \in [-3.7, -1.7]$ and $[2.8, 5.1]$. The minimum bias trigger decision was based on time-domain signal arrival time cuts which filter out most of the beam induced background such as beam-gas collisions. The PMT high voltage gain setup and signal detection thresholds were adjusted near the single minimum ionizing particle (MIP) limit as described in [41], with parameters adjusted run-by-run for different beam background and noise conditions. The offline calibration was done using a beam test data and cosmic muons when the LHC beam was not active. The detector simulations were done with run anchored setups. In the offline analysis, background and noise sensitivity were studied by requiring also a minimum charge threshold for the signal decision, in addition to more stringent time windows.

The AD detector [42] is a forward shower counter built on similar technology as the VZERO detector but with different geometry and optical coupling. The geometric acceptances over pseudorapidity are $\eta \in [-7.0, -4.9]$ and $[4.8, 6.3]$. It consists of two longitudinal (z-axis) layers of scintillating plastic with four radial transverse quadrants around the beam pipe, giving 8 channels per rapidity side. The scintillation light is steered through a wavelength shifter, to optimize the light transport which is propagated through one meter of optical fibers to Hamamatsu photomultipliers. The front-end electronics is the same as in the VZERO case and the digitized output gives hit time and accumulated charge information. The detector PMT gain and signal threshold setup was similar as with the VZERO and a signal coincidence between adjacent layers was required for the trigger. In the same way as with the VZERO, in the offline analysis, we studied the noise and background by

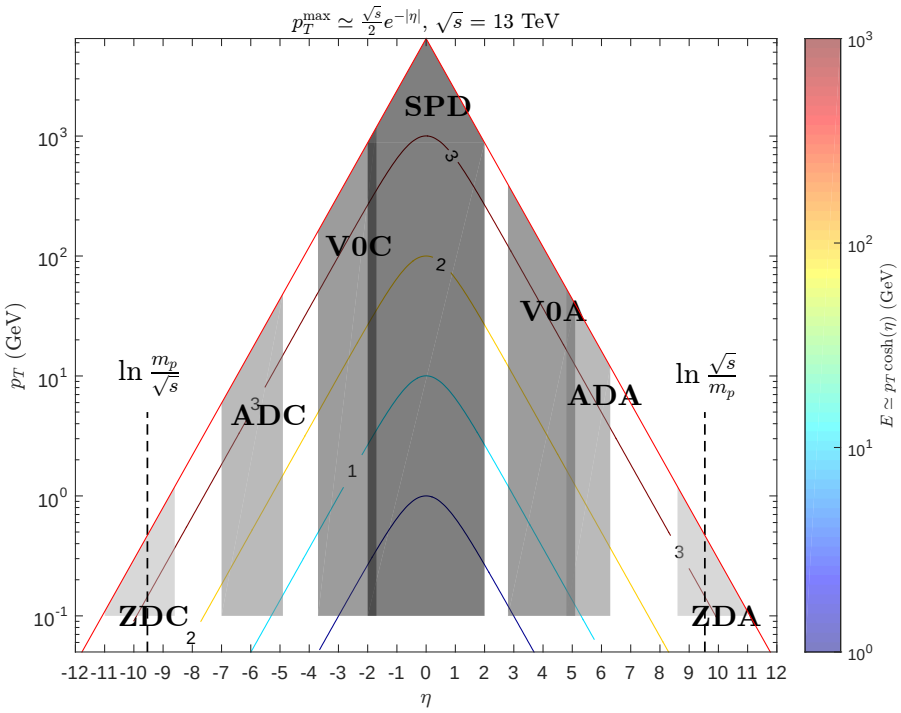

Figure 3.1: The geometric acceptance of ALICE sub-detectors. The triangle plot style adapted and extended from R. Orava.

more stringent accumulated charge and time window cuts.

The SPD (Silicon Pixel Detector) [43] of the inner tracking system (ITS) consists of two layers of pn-type silicon pixel diodes next to the interaction point with the detector cylinder radius 3.9 cm and 7.6 cm and geometric acceptance $|\eta| < 2$ and $|\eta < 1.5$. The material thickness of each pixel is 200 μm. The inner layer has 40 half-stave modules and the outer layer 80, each having 10 readout chips with 8192 pixels per chip. This gives in total 1200 readout chips with 9.83 million pixels with 50×425 μm^2 pitch size. The radiation damaged or otherwise faulty, inactive and noisy outlier chips were explicitly inspected from the data and bit masked also from the simulations. The low level online trigger implemented by the SPD is based on a Fast-OR decision where in each chip, the discriminator output of pixel cells is collected and algebraic OR is taken. In the offline analysis, at least one Fast-OR in the inner or outer SPD layer was required and the noise sensitivity was inspected by requiring more stringent criteria, such as hits in both layers. This, however, effectively also reduces

the acceptance to $|\eta| < 1.5$. Moving further in the transverse direction, there are also two layers of the silicon strip detector (SSD) and two outermost layers of the silicon drift detector (SDD) with acceptance $|\eta| < 0.9$. These layers we did not use due to their more limited acceptance. In the analysis, we used the fired chip level information. In the LHC run III, the inner tracking system will be replaced with a new silicon tracker with 12.5 gigapixels. In addition, we studied the ZDC (zero degree calorimeter) data, which however is not used in the analysis due to lacking calibration, but is used only as 'spectator' detector for general data quality cross checks. We illustrate the detector phase-space cartoon coverage in Figure 3.1.

In this analysis, three special low luminosity runs at $\sqrt{s} = 13$ TeV with identification numbers 274593, 274594 and 274595 were chosen with approximately 487k, 418k, 775k triggered minimum bias beam-beam events per run with results given here for the last one, two other used for the irreducible run-by-run uncertainties encapsulating unknown systematic uncertainties. Using the standard coding, the LHC filling scheme was

$$\begin{aligned} &\text{<SPACING>-<NBb>-<IP1/5>-<IP2>-<IP8>-<code>} \\ &= \text{Multi-57b-56b-25-20-24-4bpi-15inj}, \end{aligned} \tag{3.1}$$

meaning total 57 and 56 bunches circulating in the clockwise and counterclockwise, out of which 20 colliding bunch pairs in the ALICE IP2 interaction point. This type of beam configuration removes any off-time pileup. The instantaneous luminosity with these runs was very low, with Poisson $\mu \sim 10^{-3}$, given by the normalized global OR trigger rate R

$$\mu = -\ln(1-R), \quad \text{where} \quad R = \frac{\text{L0b}}{N_b\, f_{rev}} = \frac{\text{L0b}}{\text{LMb}} \in [0,1), \tag{3.2}$$

where the LHC revolution frequency is $f_{rev} = 11.245$ kHz, N_b is the number of interacting bunch pairs and L0b (LMb) are the global OR and bunch cross trigger frequencies. The numbers in these runs give vanishing pileup corrections, which however were part of our correction algorithms through our Möbius inversion technique [44]. For the trigger details and rates, see Appendix A.4.

A van der Meer scan using the VZERO detector was used to calibrate the absolute luminosity, details and uncertainties of this are given in [45]. The basic idea is that the VZERO_C and VZERO_A, together with their boolean AND trigger combination, provide a visible inelastic cross section

$$\sigma_{\text{V0-AND}} = 57.8 \pm 1.2 \ \text{ mb} \tag{3.3}$$

which is used to directly scale the event counts to visible cross sections because VZERO was part of our detector setup. The *visible* inelastic cross section seen by the VZERO is not a *fiducial* cross section, or neither a *total* inelastic cross section, to emphasize. These three different types of definitions were taken into account by the unfolding and extrapolation procedures.

3.2 Experimental selections and corrections

The offline signal selections are listed in Table 3.1. No high level reconstructed observables were used or are available in ALICE beyond $|\eta| < 0.9$, which would naturally be the optimal case with hypothetical large forward acceptance tracking and calorimetry.

Detector	Signal Cut
AD_C	$\Delta t \in [63, 69]$ ns
$\mathrm{V0}_C$	$\Delta t \in [0, 6]$ ns
SPD_C	$F \geq 2$
SPD_A	$F \geq 2$
$\mathrm{V0}_A$	$\Delta t \in [7, 14]$ ns
AD_A	$\Delta t \in [54, 60]$ ns

Table 3.1: Offline signal selection quality cuts in terms of the signal arrival time Δt and the number of fired chips F.

After the low-level signal decision criteria, the data was corrected for the irreducible residual beam gas, satellite collisions and noisy events. That is, diffractive events, especially single diffractive, are experimentally very similar to a fixed target like beam gas collisions passing through the time window and charge threshold cuts. To correct this, we used beam-beam, beam-empty, empty-beam and empty-empty trigger masks which were based utilizing the normal and special LHC bunch bucket sequences. The data was corrected with

$$N_j \leftarrow N_j^{(B)} - \alpha^{(A)} N_j^{(A)} - \alpha^{(C)} N_j^{(C)} + 2\alpha^{(E)} N_j^{(E)}, \tag{3.4}$$

where $N_j^{(k)}$ represent the number of events in the j-th combination with the k running over the aforementioned special bunch trigger collected event statistics. The last factor with a positive sign and a factor of two takes into account the double counting. The correction scale factors $\alpha^{(k)}$ taking into account the different bunch sequence luminosity differences and the deterministic and random trigger downscaling were obtained from the trigger statistics with

$$\alpha^{(k)} = \left[\frac{\mathrm{LMb}(B)}{\mathrm{LMb}(k)}\right] \left[\frac{\mathrm{L0a}(B)/\mathrm{L0b}(B)}{\mathrm{L0a}(k)/\mathrm{L0b}(k)}\right], \tag{3.5}$$

where $k = A$ is a beam from the A-side and nothing from the C-side, $k = C$ is a beam from the C-side and nothing from the A-side and $k = E$ is nothing from both

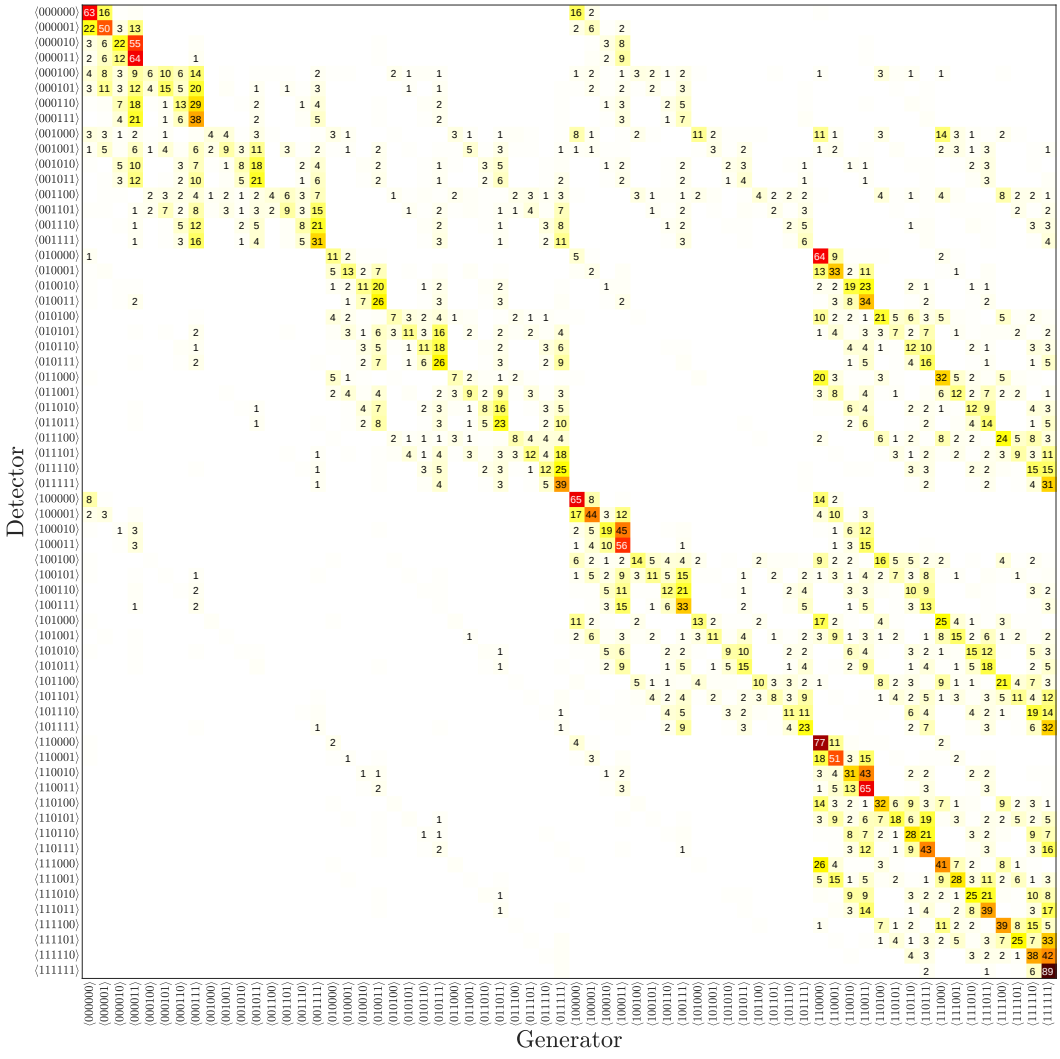

Figure 3.2: ALICE simulation at $\sqrt{s} = 13$ TeV: The 6-dimensional detector folding matrix constructed with Pythia 6 MC + GEANT. The matrix elements are probabilities (%) with each column normalized to unity. Elements $< 1\%$ are not shown.

sides. The $k = E$ case is negligible whenever detectors are operating with low noise levels. The trigger statistics are given in Tables A.3, A.4 and A.5. The detector level marginal distributions of each detector are given in Figures A.1 to A.13.

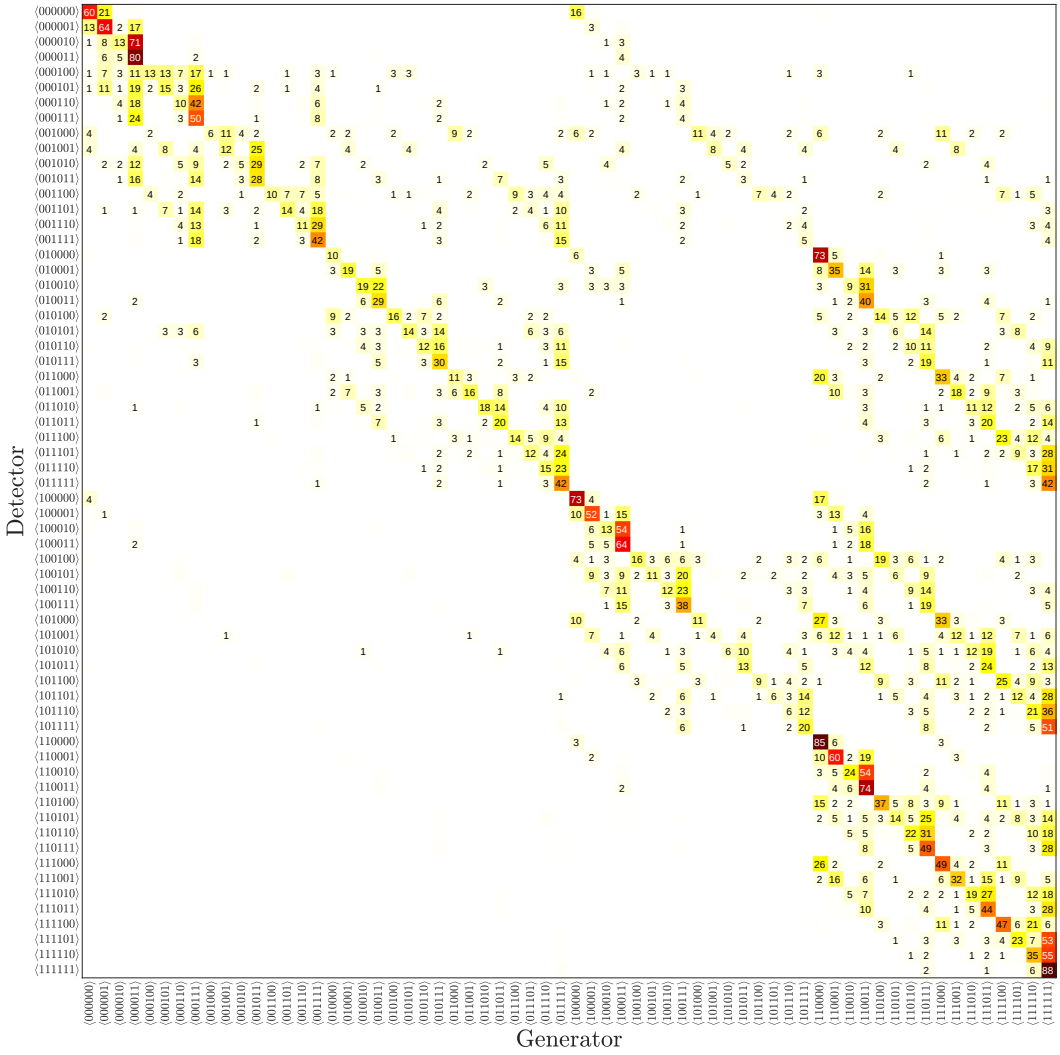

Figure 3.3: ALICE simulation at $\sqrt{s} = 13$ TeV: The 6-dimensional detector folding matrix constructed with Phojet MC + GEANT, otherwise the same as Figure 3.2.

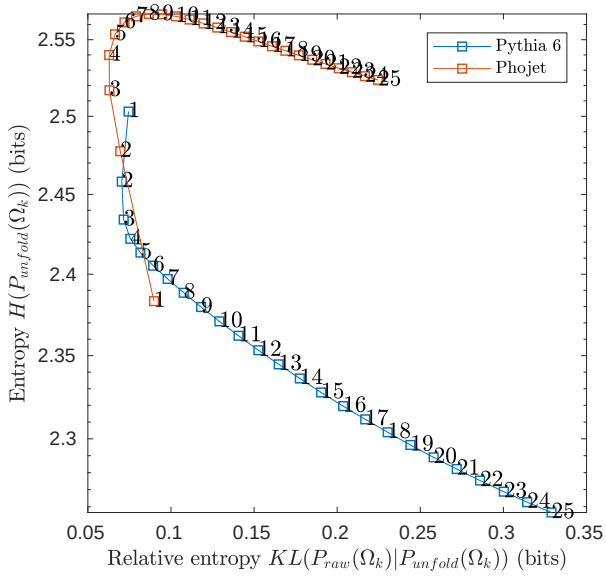

Figure 3.4: ALICE data at $\sqrt{s} = 13$ TeV: The unfolded data Shannon entropy (vertical) versus relative entropy of raw data vs the unfolded (horizontal). The evolution is shown using two MC models used in the detector response simulation. A suitable number of unfolding algorithm iterations ($1 \ldots 25$) is near the maximum curvature.

Now speaking of 6-dimensional unfolding of data, we used the standard iterative Bayes method (D'Agostini) of the RooUnfold package [46]. However, we compared the implementation against our own implementation of the same algorithm and also Tikhonov regularized matrix inversion. To be precise, this iterative Bayes unfolding is a frequentist maximum likelihood method using the Expectation Maximization algorithm, which uses the Bayes formula, but is not a Bayesian method. The only Bayesian part is actually the number of iterations, which is an implicit regulator or prior. The unfolding here inverts the inefficiencies and the smearing flow of event topologies from one combinatorial category to another one, thus formally easily described by a folding matrix in the same way as any single dimensional histogram unfolded measurement. These combinatorial folding matrices are shown in Figures 3.2 and 3.3, which demonstrate the major role of detector material rescattering and efficiency corrections.

In the forward domain, even half of the accumulated charge seen by the detector

is from particles propagating from the material interactions, such as $\gamma \to e^+e^-$ conversions. This is demonstrated in Figure A.17. Fiducial (generator) level event topologies are easily transformed into another topology at the detector level. The simplest acceptance characterizations, here for the visualization purposes, are the diffractive mass efficiency $\times$ acceptances shown in Figure A.16 and A.15, where we see that for the SPD the pure fiducial acceptance and the efficiency $\times$ acceptance are very close. However, for the AD we see a large difference between these two. This is due to the material distributions which actually yield larger effective acceptance than what is geometrically expected. The difference between Pythia 6 and Phojet is due to differences in the fragmentation of the diffractive systems. Phojet has a hard (perturbative) p_t-tail, Pythia 6 generates diffractive events only with soft p_t. Also, the multiplicity distributions are different, as is visible in Figure A.14 where the detector level MC/Data ratios are shown. For a generator level comparisons together with Pythia 8, see the work in [47].

Figure 3.4 demonstrates the unfolding regularization parameter (number of iterations) abstract behavior in terms of Shannon entropy and relative entropy (Kullback-Leibler divergence) between the raw with no unfolding and the unfolded measurement. Interestingly, the two different MC models have approximately the same 'sweet spot' but opposite trajectories in this abstract entropy space. We found out that this technique gives a purely data driven handle to control the regularization strength. The point where the regularization should be optimal is the point of maximum curvature, by highly often used L-curve heuristics in inverse problems [48]. Or it could be also the point where trajectories between different models cross. Zero iterations gives no unfolding, whereas too many iterations will push the results towards Monte Carlo estimates, speaking in general. Based on this and simulations, we chose the point of 4 ± 2 iterations in the unfolding algorithm. Accidentally, the default values of the LHC unfolding package parameters are usually around 4 iterations. We should study this interesting topic further in the future, because it has been basically neglected in all existing rapidity gap based measurements, even unfolded ones, because they integrate out the multidimensional information.

We found out that the unfolding has here the largest experimental systematic uncertainty besides the forward detector geometry and material budget simulation, the SPD chip noise and the V0 and AD signal response modeling uncertainties. Understanding these systematic uncertainties in extreme detail would be required for higher precision measurement. Also the beam-gas correction can be in some runs very large, but our combinatorics has a self-consistent way to correct it. Still, run-by-run differences remain much larger than statistical uncertainties in some combinations. A forward tracking and calorimetry with very low material budget would be beneficial.

3.3 Unfolded fiducial partial cross sections

The 6D-measurement fiducial definition we use is charged particles within

$$\eta \in [-7.0, -4.9], [-3.7, -1.7], [-2.0, 0], [0, 2.0], [2.8, 5.1], [4.8, 6.3] \\ \wedge \ p_t > 0.05 \text{ GeV}. \tag{3.6}$$

This corresponds to the nominal acceptance of the AD, VZERO and SPD detectors. In principle, one could have fine-tuned and optimized the fiducial definition of the measurement beyond the geometric acceptances by minimizing the off-diagonality of the folding matrix. That is, by shifting the simulated fiducial box boundaries per acceptance slice and then re-running the matrix construction, checking the off-diagonality and re-iterating. In the same way, one could also decide if only charged or both charged + neutral particles should be taken as the fiducial definition. However, here we took simply the geometric η-boundaries with a 'nominal' p_t-cutoff for charged particles. To denote the 6D-acceptance domains, we use binary coding with six bits.

In Figures 3.5 and 3.6 we have the main result, the unfolded combinatorial partial fiducial cross sections, compared with different minimum bias Monte Carlo event generators. The pink uncertainty bands represent total systematic uncertainty obtained by summing the individual sources in quadrature. The systematic uncertainty sources include the van der Meer scan luminosity calibration, the detector unfolding uncertainties and run-by-run variations reflecting detector calibration and simulation mismatches. The numerical values are given in Tables 3.2, 3.3, 3.4 and 3.5. Now when comparing with Monte Carlo models, the biggest differences are obtained with very forward low mass diffractive like combinations, such as $\langle 010000 \rangle$, which have also the largest uncertainty. Numerous low level cut variations in terms of the accumulated charge and time domain response were inspected, without any major difference. We see that this measurement should provide a highly systematic way to improve the event generator modeling and tunes, together with all the other existing soft QCD measurements, naturally. We emphasize that our definition of diffraction here is well-posed, we do not need to know any internal details about the event generators, only the final state information is needed for the comparison. Thus, this type of measurement is fully compatible with the increasingly popular Rivet [26] measurement-theory comparison framework requirements.

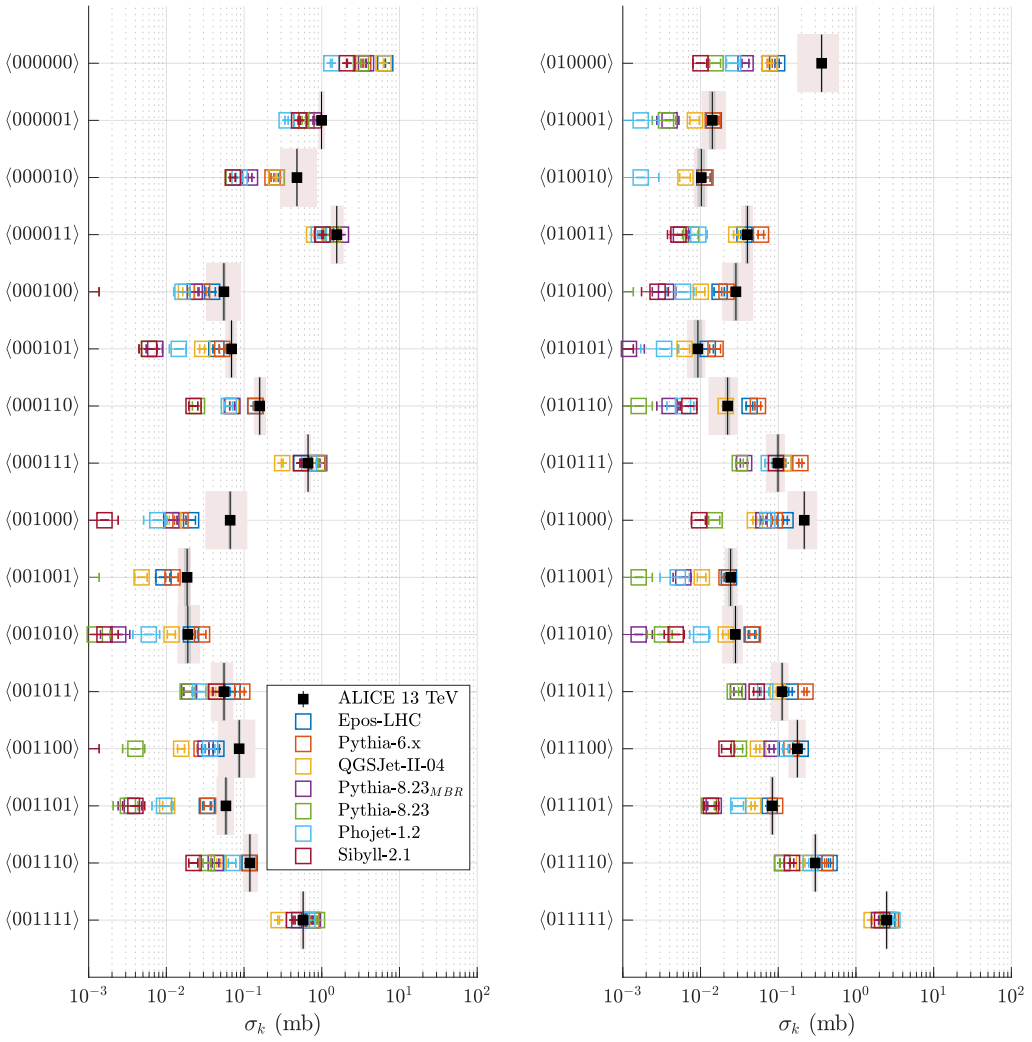

Figure 3.5: ALICE data at $\sqrt{s} = 13$ TeV and MC event generators: The fiducial partial cross sections [0-15] and [16-31].

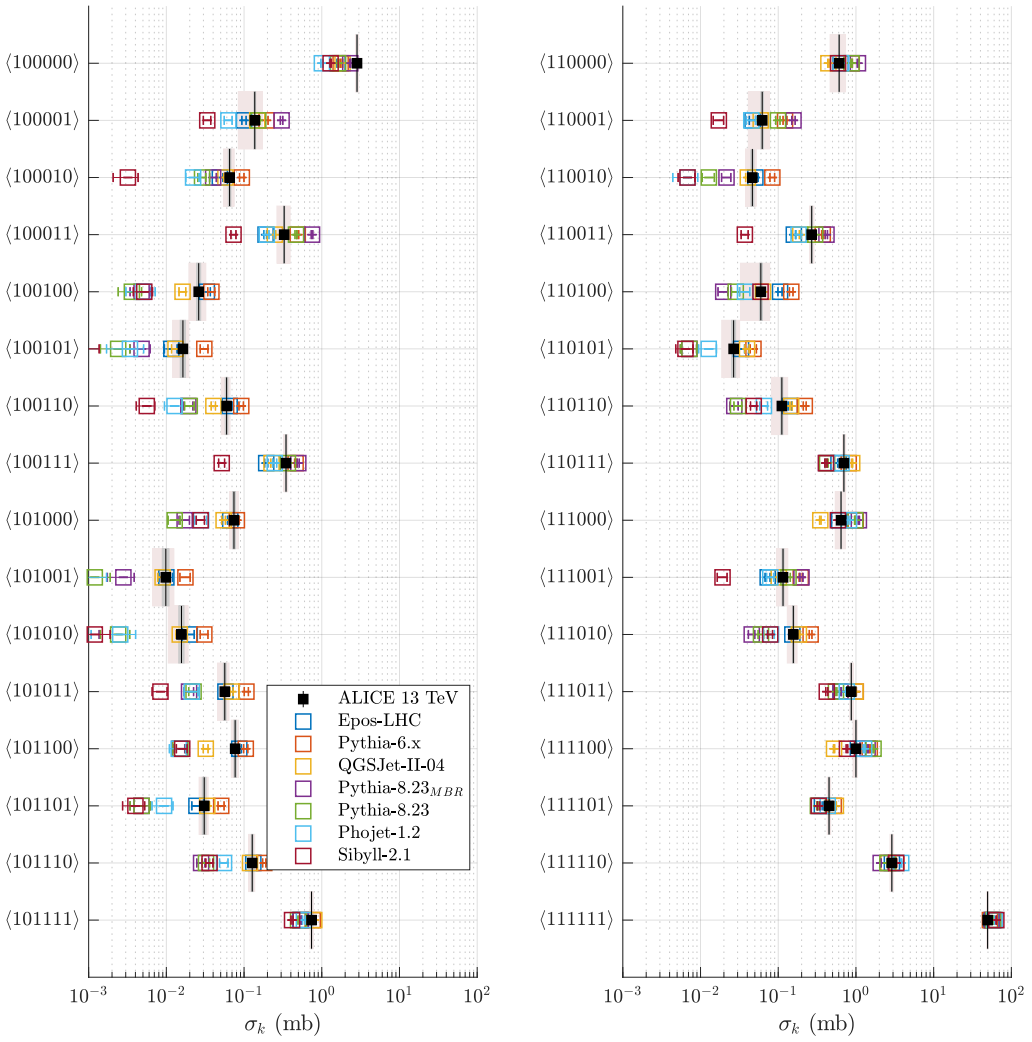

Figure 3.6: ALICE data at $\sqrt{s} = 13$ TeV and MC event generators: The fiducial partial cross sections [32-47] and [48-63].

	X-Section	value (mb)	stat	tot.syst	lumi	unfold	run-by-run
0	$\langle 000000 \rangle$	0.000	± 0.000	$^{+0.000}_{-0.000}$	± 0.000	$^{+0.000}_{-0.000}$	$^{+0.000}_{-0.000}$
1	$\langle 000001 \rangle$	0.992	± 0.010	$^{+0.092}_{-0.075}$	± 0.021	$^{+0.056}_{-0.036}$	$^{+0.070}_{-0.062}$
2	$\langle 000010 \rangle$	0.479	± 0.007	$^{+0.375}_{-0.181}$	± 0.010	$^{+0.095}_{-0.090}$	$^{+0.363}_{-0.157}$
3	$\langle 000011 \rangle$	1.558	± 0.012	$^{+0.325}_{-0.239}$	± 0.032	$^{+0.047}_{-0.021}$	$^{+0.320}_{-0.236}$
4	$\langle 000100 \rangle$	0.055	± 0.002	$^{+0.034}_{-0.022}$	± 0.001	$^{+0.014}_{-0.015}$	$^{+0.031}_{-0.016}$
5	$\langle 000101 \rangle$	0.069	± 0.003	$^{+0.010}_{-0.011}$	± 0.001	$^{+0.004}_{-0.007}$	$^{+0.009}_{-0.008}$
6	$\langle 000110 \rangle$	0.159	± 0.004	$^{+0.030}_{-0.023}$	± 0.003	$^{+0.012}_{-0.009}$	$^{+0.027}_{-0.021}$
7	$\langle 000111 \rangle$	0.666	± 0.008	$^{+0.047}_{-0.066}$	± 0.014	$^{+0.044}_{-0.064}$	$^{+0.010}_{-0.010}$
8	$\langle 001000 \rangle$	0.066	± 0.003	$^{+0.042}_{-0.034}$	± 0.001	$^{+0.035}_{-0.031}$	$^{+0.023}_{-0.014}$
9	$\langle 001001 \rangle$	0.019	± 0.001	$^{+0.002}_{-0.004}$	± 0.000	$^{+0.002}_{-0.004}$	$^{+0.000}_{-0.000}$
10	$\langle 001010 \rangle$	0.019	± 0.001	$^{+0.008}_{-0.005}$	± 0.000	$^{+0.002}_{-0.002}$	$^{+0.007}_{-0.004}$
11	$\langle 001011 \rangle$	0.055	± 0.002	$^{+0.017}_{-0.017}$	± 0.001	$^{+0.011}_{-0.015}$	$^{+0.013}_{-0.009}$
12	$\langle 001100 \rangle$	0.087	± 0.003	$^{+0.051}_{-0.040}$	± 0.002	$^{+0.038}_{-0.034}$	$^{+0.034}_{-0.020}$
13	$\langle 001101 \rangle$	0.058	± 0.002	$^{+0.014}_{-0.014}$	± 0.001	$^{+0.004}_{-0.009}$	$^{+0.014}_{-0.010}$
14	$\langle 001110 \rangle$	0.119	± 0.003	$^{+0.029}_{-0.021}$	± 0.002	$^{+0.006}_{-0.005}$	$^{+0.028}_{-0.020}$
15	$\langle 001111 \rangle$	0.576	± 0.008	$^{+0.037}_{-0.043}$	± 0.012	$^{+0.019}_{-0.032}$	$^{+0.029}_{-0.027}$

Table 3.2: ALICE data at $\sqrt{s} = 13$ TeV: The fiducial partial cross sections [0,15].

	X-Section	value (mb)	stat	tot.syst	lumi	unfold	run-by-run
16	$\langle 010000 \rangle$	0.362	± 0.006	$^{+0.234}_{-0.184}$	± 0.008	$^{+0.168}_{-0.159}$	$^{+0.163}_{-0.092}$
17	$\langle 010001 \rangle$	0.014	± 0.001	$^{+0.007}_{-0.004}$	± 0.000	$^{+0.000}_{-0.000}$	$^{+0.007}_{-0.004}$
18	$\langle 010010 \rangle$	0.010	± 0.001	$^{+0.002}_{-0.002}$	± 0.000	$^{+0.001}_{-0.001}$	$^{+0.002}_{-0.001}$
19	$\langle 010011 \rangle$	0.040	± 0.002	$^{+0.006}_{-0.006}$	± 0.001	$^{+0.002}_{-0.004}$	$^{+0.006}_{-0.005}$
20	$\langle 010100 \rangle$	0.029	± 0.002	$^{+0.018}_{-0.009}$	± 0.001	$^{+0.001}_{-0.003}$	$^{+0.018}_{-0.009}$
21	$\langle 010101 \rangle$	0.009	± 0.001	$^{+0.002}_{-0.002}$	± 0.000	$^{+0.002}_{-0.002}$	$^{+0.001}_{-0.001}$
22	$\langle 010110 \rangle$	0.022	± 0.001	$^{+0.007}_{-0.009}$	± 0.000	$^{+0.006}_{-0.009}$	$^{+0.005}_{-0.004}$
23	$\langle 010111 \rangle$	0.099	± 0.003	$^{+0.020}_{-0.029}$	± 0.002	$^{+0.019}_{-0.028}$	$^{+0.007}_{-0.006}$
24	$\langle 011000 \rangle$	0.216	± 0.005	$^{+0.095}_{-0.083}$	± 0.004	$^{+0.067}_{-0.070}$	$^{+0.068}_{-0.044}$
25	$\langle 011001 \rangle$	0.024	± 0.002	$^{+0.005}_{-0.004}$	± 0.001	$^{+0.002}_{-0.001}$	$^{+0.005}_{-0.004}$
26	$\langle 011010 \rangle$	0.028	± 0.002	$^{+0.006}_{-0.009}$	± 0.001	$^{+0.006}_{-0.009}$	$^{+0.000}_{-0.000}$
27	$\langle 011011 \rangle$	0.112	± 0.003	$^{+0.021}_{-0.030}$	± 0.002	$^{+0.021}_{-0.030}$	$^{+0.005}_{-0.005}$
28	$\langle 011100 \rangle$	0.176	± 0.004	$^{+0.046}_{-0.039}$	± 0.004	$^{+0.024}_{-0.026}$	$^{+0.039}_{-0.028}$
29	$\langle 011101 \rangle$	0.084	± 0.003	$^{+0.007}_{-0.006}$	± 0.002	$^{+0.004}_{-0.003}$	$^{+0.005}_{-0.005}$
30	$\langle 011110 \rangle$	0.300	± 0.005	$^{+0.021}_{-0.025}$	± 0.006	$^{+0.017}_{-0.022}$	$^{+0.010}_{-0.010}$
31	$\langle 011111 \rangle$	2.484	± 0.016	$^{+0.102}_{-0.095}$	± 0.052	$^{+0.024}_{-0.004}$	$^{+0.084}_{-0.080}$

Table 3.3: ALICE data at $\sqrt{s} = 13$ TeV: The fiducial partial cross sections [16,31].

	X-Section	value (mb)	stat	tot.syst	lumi	unfold	run-by-run
32	$\langle 100000 \rangle$	2.842	± 0.017	$^{+0.133}_{-0.126}$	± 0.059	$^{+0.024}_{-0.026}$	$^{+0.117}_{-0.109}$
33	$\langle 100001 \rangle$	0.137	± 0.004	$^{+0.035}_{-0.053}$	± 0.003	$^{+0.033}_{-0.052}$	$^{+0.012}_{-0.011}$
34	$\langle 100010 \rangle$	0.065	± 0.003	$^{+0.010}_{-0.011}$	± 0.001	$^{+0.002}_{-0.008}$	$^{+0.009}_{-0.007}$
35	$\langle 100011 \rangle$	0.328	± 0.006	$^{+0.064}_{-0.063}$	± 0.007	$^{+0.015}_{-0.042}$	$^{+0.062}_{-0.046}$
36	$\langle 100100 \rangle$	0.026	± 0.002	$^{+0.006}_{-0.007}$	± 0.001	$^{+0.004}_{-0.006}$	$^{+0.004}_{-0.003}$
37	$\langle 100101 \rangle$	0.016	± 0.001	$^{+0.003}_{-0.004}$	± 0.000	$^{+0.003}_{-0.004}$	$^{+0.001}_{-0.001}$
38	$\langle 100110 \rangle$	0.060	± 0.002	$^{+0.006}_{-0.009}$	± 0.001	$^{+0.004}_{-0.008}$	$^{+0.005}_{-0.004}$
39	$\langle 100111 \rangle$	0.347	± 0.006	$^{+0.018}_{-0.021}$	± 0.007	$^{+0.003}_{-0.012}$	$^{+0.017}_{-0.015}$
40	$\langle 101000 \rangle$	0.074	± 0.003	$^{+0.010}_{-0.009}$	± 0.002	$^{+0.009}_{-0.007}$	$^{+0.005}_{-0.005}$
41	$\langle 101001 \rangle$	0.010	± 0.001	$^{+0.003}_{-0.003}$	± 0.000	$^{+0.002}_{-0.003}$	$^{+0.002}_{-0.001}$
42	$\langle 101010 \rangle$	0.016	± 0.001	$^{+0.003}_{-0.005}$	± 0.000	$^{+0.003}_{-0.005}$	$^{+0.001}_{-0.001}$
43	$\langle 101011 \rangle$	0.057	± 0.002	$^{+0.007}_{-0.011}$	± 0.001	$^{+0.007}_{-0.011}$	$^{+0.002}_{-0.002}$
44	$\langle 101100 \rangle$	0.077	± 0.003	$^{+0.008}_{-0.008}$	± 0.002	$^{+0.007}_{-0.007}$	$^{+0.003}_{-0.003}$
45	$\langle 101101 \rangle$	0.031	± 0.002	$^{+0.004}_{-0.004}$	± 0.001	$^{+0.003}_{-0.004}$	$^{+0.002}_{-0.002}$
46	$\langle 101110 \rangle$	0.127	± 0.004	$^{+0.010}_{-0.012}$	± 0.003	$^{+0.009}_{-0.012}$	$^{+0.003}_{-0.003}$
47	$\langle 101111 \rangle$	0.738	± 0.009	$^{+0.040}_{-0.035}$	± 0.015	$^{+0.030}_{-0.023}$	$^{+0.022}_{-0.021}$

Table 3.4: ALICE data at $\sqrt{s} = 13$ TeV: The fiducial partial cross sections [32,47].

	X-Section	value (mb)	stat	tot.syst	lumi	unfold	run-by-run
48	$\langle 110000 \rangle$	0.608	± 0.008	$^{+0.126}_{-0.143}$	± 0.013	$^{+0.119}_{-0.138}$	$^{+0.040}_{-0.036}$
49	$\langle 110001 \rangle$	0.062	± 0.002	$^{+0.015}_{-0.021}$	± 0.001	$^{+0.012}_{-0.020}$	$^{+0.008}_{-0.007}$
50	$\langle 110010 \rangle$	0.046	± 0.002	$^{+0.006}_{-0.009}$	± 0.001	$^{+0.004}_{-0.008}$	$^{+0.004}_{-0.003}$
51	$\langle 110011 \rangle$	0.270	± 0.005	$^{+0.018}_{-0.016}$	± 0.006	$^{+0.009}_{-0.006}$	$^{+0.015}_{-0.014}$
52	$\langle 110100 \rangle$	0.060	± 0.002	$^{+0.018}_{-0.027}$	± 0.001	$^{+0.017}_{-0.027}$	$^{+0.006}_{-0.005}$
53	$\langle 110101 \rangle$	0.027	± 0.002	$^{+0.005}_{-0.008}$	± 0.001	$^{+0.005}_{-0.008}$	$^{+0.001}_{-0.001}$
54	$\langle 110110 \rangle$	0.111	± 0.003	$^{+0.021}_{-0.029}$	± 0.002	$^{+0.019}_{-0.028}$	$^{+0.008}_{-0.007}$
55	$\langle 110111 \rangle$	0.700	± 0.008	$^{+0.037}_{-0.026}$	± 0.015	$^{+0.029}_{-0.015}$	$^{+0.017}_{-0.016}$
56	$\langle 111000 \rangle$	0.643	± 0.008	$^{+0.087}_{-0.101}$	± 0.013	$^{+0.063}_{-0.086}$	$^{+0.058}_{-0.050}$
57	$\langle 111001 \rangle$	0.115	± 0.003	$^{+0.018}_{-0.020}$	± 0.002	$^{+0.008}_{-0.015}$	$^{+0.015}_{-0.012}$
58	$\langle 111010 \rangle$	0.156	± 0.004	$^{+0.018}_{-0.026}$	± 0.003	$^{+0.017}_{-0.025}$	$^{+0.005}_{-0.004}$
59	$\langle 111011 \rangle$	0.871	± 0.009	$^{+0.053}_{-0.039}$	± 0.018	$^{+0.047}_{-0.030}$	$^{+0.017}_{-0.017}$
60	$\langle 111100 \rangle$	1.000	± 0.010	$^{+0.046}_{-0.082}$	± 0.021	$^{+0.032}_{-0.076}$	$^{+0.024}_{-0.023}$
61	$\langle 111101 \rangle$	0.451	± 0.007	$^{+0.022}_{-0.023}$	± 0.009	$^{+0.001}_{-0.010}$	$^{+0.019}_{-0.018}$
62	$\langle 111110 \rangle$	2.897	± 0.017	$^{+0.155}_{-0.153}$	± 0.060	$^{+0.041}_{-0.062}$	$^{+0.137}_{-0.126}$
63	$\langle 111111 \rangle$	49.444	± 0.070	$^{+1.272}_{-1.319}$	± 1.027	$^{+0.108}_{-0.404}$	$^{+0.743}_{-0.724}$

Table 3.5: ALICE data at $\sqrt{s} = 13$ TeV: The fiducial partial cross sections [48,63].

3.4 Fiducial and total inelastic cross section

The fiducial inelastic measurement and the corresponding extrapolated full phase space total inelastic cross section are given numerically in Table 3.6 and visually in Figure 3.7. Figure 3.8 shows in detail the Monte Carlo model based total inelastic cross section extrapolations based on the measured partial cross sections. EPOS-LHC and QGSJet-II-04 based extrapolations give values with a closest match with respect to the total inelastic measurement done by the TOTEM experiment at $\sqrt{s} = 13$ TeV giving 79.5 ± 1.8 mb, which is based on the optical theorem and the elastic scattering with Mandelstam $t \to 0$ extrapolation [49]. That measurement has the smallest experimental uncertainty at the LHC. Alternatively, one should instrument the experiments with extremely forward shower counters to observe the total inelastic cross section directly.

Cross Section	value $\pm$ (stat+syst)
σ_{inel}^{fid}	71.4 ± 1.8 mb
σ_{inel}^{tot} (ext.)	77.3 ± 2.1 mb

Table 3.6: ALICE data at $\sqrt{s} = 13$ TeV: The inelastic cross section in the fiducial domain and the total extrapolated one obtained using re-fitted Pythia 6.

The EPOS-LHC, QGSJet-II-04 and Pythia 6 generators have here the best overall tune, in terms of χ^2. The χ^2-value is simply calculated against the total inelastic re-scaled Monte Carlo estimate of the fiducial partial cross sections and the measured ones, that is, one number is being varied in the fit. We took care of the technical scale bias easily introduced in χ^2 type fits. All of the event generators have some obvious tension given that the number of degrees of freedom is here $2^N - 2 = 62$. This global tension is easily explained by certain specific deviating partial cross sections. In Figure 3.7 we have also our own re-tune of Pythia 6 based extrapolation indicated, based on the extracted diffractive cross sections and the extracted soft Pomeron intercept value. Our tune is significantly closer to the EPOS-LHC and QGSJet-II-04 than the original Pythia 6 in the CRMC package.

The overall inelastic fiducial acceptances $\sigma_{inel}^{fid}/\sigma_{inel}^{tot}$ predicted by different generators are: EPOS-LHC (0.92), QGSJet-II-04 (0.92), Pythia 8.23-MBR (0.95), Pythia 6 (0.96), Pythia 8.23 (0.96), Sibyll 2.1 (0.97) and Phojet 1.2 (0.98). This indicates that all other event generators than EPOS and QGSJet-II-04 clearly underestimate the cross section of the invisible low mass diffraction. This conclusion is compatible with our other results, such as the F^*-projected gap distributions in Section 3.6.

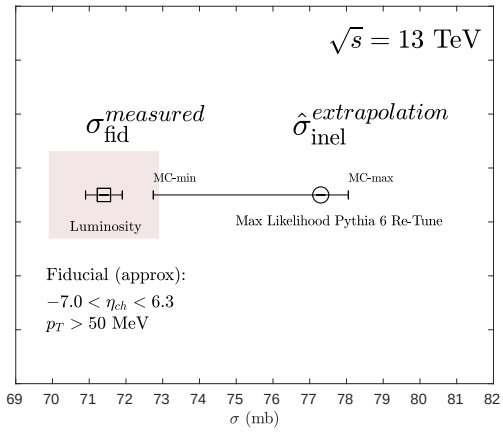

Figure 3.7: ALICE data at $\sqrt{s} = 13$ TeV: The fiducial inelastic proton-proton cross section on the left, and the extrapolated total inelastic cross section using different MC models on the right. The luminosity uncertainty is denoted with a pink box.

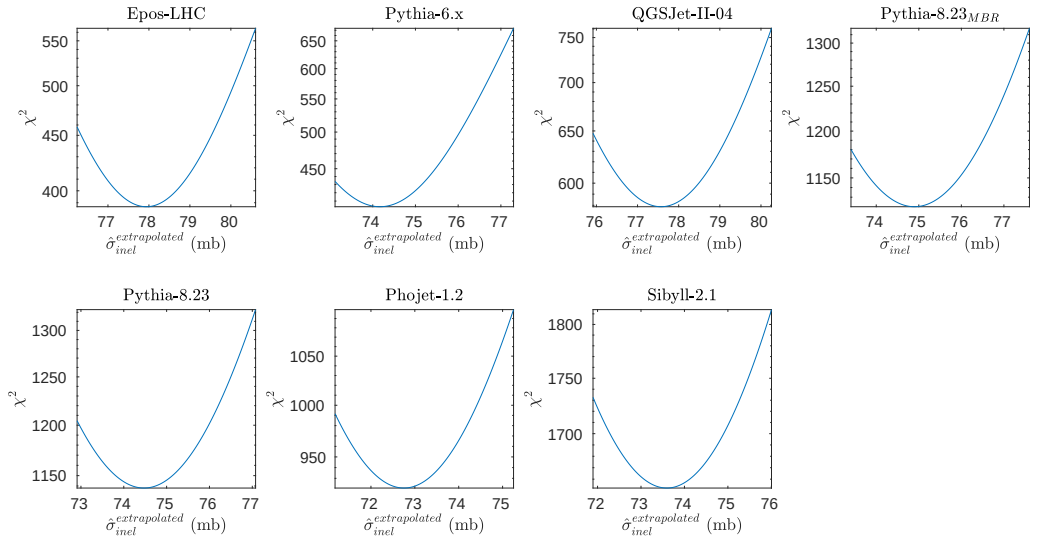

Figure 3.8: ALICE data at $\sqrt{s} = 13$ TeV: The total inelastic cross section extrapolation fit χ^2-values with different MC models using the fiducial unfolded combinatorial cross sections as the fit input.

3.5 Diffractive cross sections and the soft pomeron

Extracting the diffractive cross section is not experimentally a uniquely posed task, but it can be mathematically clearly formulated as a multidimensional probabilistic fitting or classification problem as first done in a work with CDF data by me [50], or later by Welti with TOTEM+CMS data in [51]. Also, in a work by Matthews [52] the trigger combinations were used to extract the diffractive cross sections using ALICE data. Here, we improve the methodology by a maximum likelihood formulation, which allows also simultaneously to vary the minimum average rapidity gap definition and the effective Pomeron intercept. But most importantly, here we implemented for the first time a proper multidimensional fiducial measurement which then allows to implement numerous fits with any future Monte Carlo event generator, using the unfolded partial cross sections from Tables 3.2, 3.3, 3.4 and 3.5.

In Figure 3.9 we have the diffractive cross section extractions using the multidimensional maximum likelihood 6D-fit machinery and their extrapolations to the full phase space with uncertainties defined as before. SDL(R) denotes the diffractive system to negative (positive) rapidities, DD denotes double diffractive and ND incoherent non-diffractive processes. The relative uncertainties are the largest with single diffraction, due to run-by-run variations (beam background). We have done the extractions as a function of the average kinematic rapidity gap, which is simply done by calculating the invariant mass of the diffractive system at the generator level and then calculating the equivalent gap sizes

$$\text{SD: } \langle \Delta Y \rangle_{\min} = -\ln\left(\frac{M^2}{s}\right) \tag{3.7}$$

$$\text{DD: } \langle \Delta Y \rangle_{\min} = -\ln\left(\frac{M_1^2 M_2^2}{s s_0}\right), \quad s_0 = 1\text{GeV}^2. \tag{3.8}$$

Then we excluded the simulated events with smaller gap values. Thus, one may take the traditional integration bound of $\langle \Delta Y \rangle_{\min} = 3$ and compare with previous measurements in [53]. Our definition is a sharp one, because this simply means that we have a cutoff in the diffractive model before it is fragmented. The ALICE detector has an asymmetric rapidity acceptance, thus, the fiducial single diffractive cross sections are asymmetric between forward and backward directions. Their extrapolated values are symmetric, as they should, if the detector simulation together with data is well understood. We remark here that this is by no means automatically guaranteed, given that the observables with forward shower counters are very low level, that is, no tracks and no transverse momentum being measured so the calibration of the data is non-trivial.

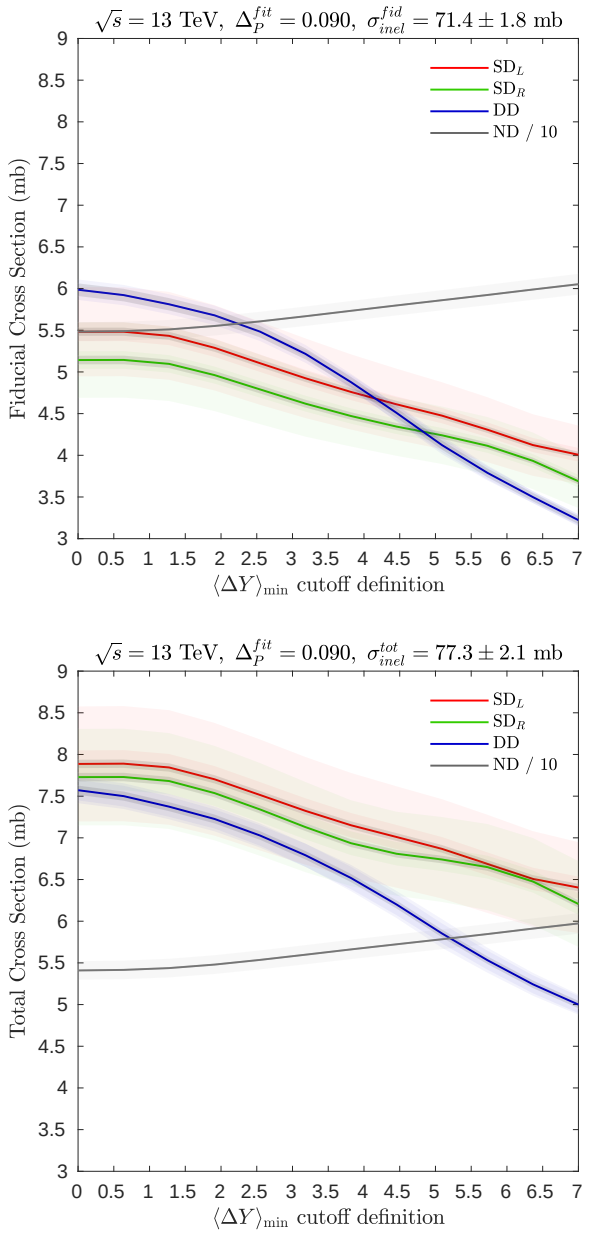

Figure 3.9: ALICE data at $\sqrt{s} = 13$ TeV: The unfolded fiducial cross sections (top) and the full phase space extrapolated (bottom) as a function of the minimum average rapidity gap cutoff based on 6D-re-fitting against Pythia 6 MC templates.

In Figure 3.10 we illustrate the typical fit cost of the soft pomeron intercept Δ_P, which is a fundamental free parameter of the Regge theory. Note that minimizing the Kullback-Leibler divergence gives a mathematically equivalent solution to the maximum likelihood, well known in the field of machine learning. Exact numerical values are given in Table 3.7 yielding a total estimate

$$\Delta_P = 0.094 \pm 0.01\,(\text{stat+syst}), \tag{3.9}$$

where we take a simple median and standard deviation of the individual extractions. This extraction is the first maximum likelihood multidimensional fit of this parameter at the LHC, as far as we know, also the most precise one. For comparison, a recent measurement by the ATLAS+ALFA experiment obtained a more uncertain value of $\Delta_P = 0.07 \pm 0.09$ in single diffraction with forward proton measurements at $\sqrt{s} = 8$ TeV [54].

The extracted intercept value is very close to the one obtained in classic fits by Donnachie and Landshoff giving the soft pomeron $\alpha(t) = 1+\Delta_P+\alpha' t \simeq 1.08+0.25t$, based on the total cross section values [55]. Actually, in later fits by DL, the intercept is a bit higher. Here, our fit is based on the differential diffractive mass distribution which we indirectly re-fit using the combinatorial cross sections as the observables and Monte Carlo model based template distributions. The pomeron intercept refit has a larger MC based diffractive mass distribution uncertainty with Pythia 6 than Phojet, because Pythia 6 includes 'fudge factors' at low and high masses, whereas Phojet has pure power law dependence. However, we note that this fit can be easily re-done with the future Monte Carlo models using the measured unfolded partial cross section values, with any fast (or brute force) fit techniques available. The soft pomeron slope α' can be extracted for example in the exclusive central diffraction, from the system transverse momentum dependence or forward proton measurements. We shall remark here that the Pomeron intercept and slope are model dependent quantities, in a more fundamental Regge calculus case one may talk about the 'bare' intercept and slope which are then dressed a bit like in field theory. These values are in principle calculable from the field theory of QCD, but in practice there no known non-perturbative techniques yet available. For more about this, see the discussion of the eikonal pomeron in Chapter 5.

MC model	Run	Δ_P pre-unfold	Δ_P post-unfold
Pythia 6	274593	0.088	0.102
Pythia 6	274594	0.094	0.106
Pythia 6	274595	0.085	0.099
Phojet	274593	0.094	0.091
Phojet	274594	0.100	0.097
Phojet	274595	0.089	0.087

Table 3.7: ALICE data at $\sqrt{s} = 13$ TeV: Run-by-run maximum likelihood 6D-fit extractions of the soft Pomeron intercept under two different MC models and before and after unfolding. Statistical uncertainties are negligible.

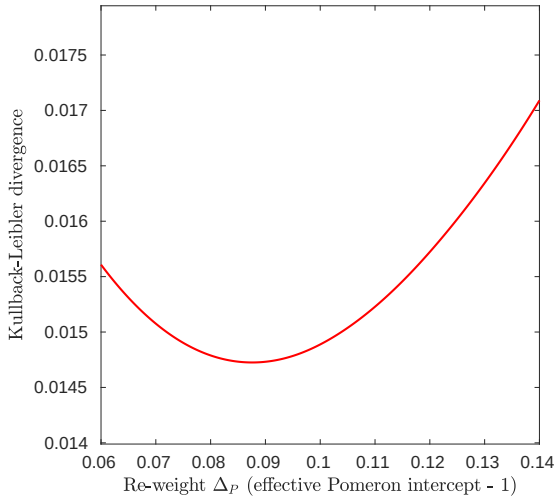

Figure 3.10: ALICE data at $\sqrt{s} = 13$ TeV: Typical behavior of the soft pomeron intercept Δ_P in the maximum likelihood 6D-fit against Pythia 6 MC templates.

3.6 F^*-projected observables

In Figures 3.12, 3.13 and 3.11 we have the unfolded pseudorapidity gap distributions projected using our F^*-projection algorithm, described in [44] and Appendix C.6. The type I boundary conditions calculate gaps with respect to the detector edge and type II between any two particle within the detector. The ALICE detector has no strict detector capabilities for direct measurement of these such as large calorimetry or tracking with p_t measurement, thus our novel MC projection algorithm based on the combinatorial cross sections, represent a state-of-the-art approach. It is not a pure measurement, but an overcomplete 'basis' projection.

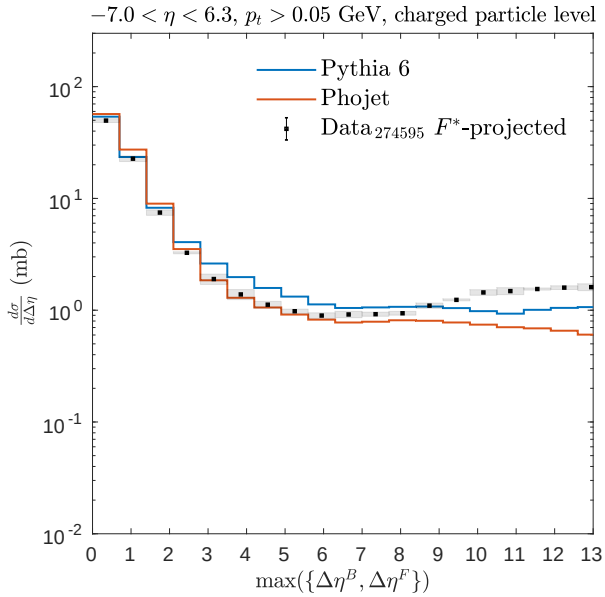

Figure 3.11: ALICE data at $\sqrt{s} = 13$ TeV and MC event generators: Type I pseudorapidity gap size distribution measured with respect to the detector edge, and the maximum of the backward *or* the forward direction gap is chosen per event. The projection uncertainties denote MC model variations.

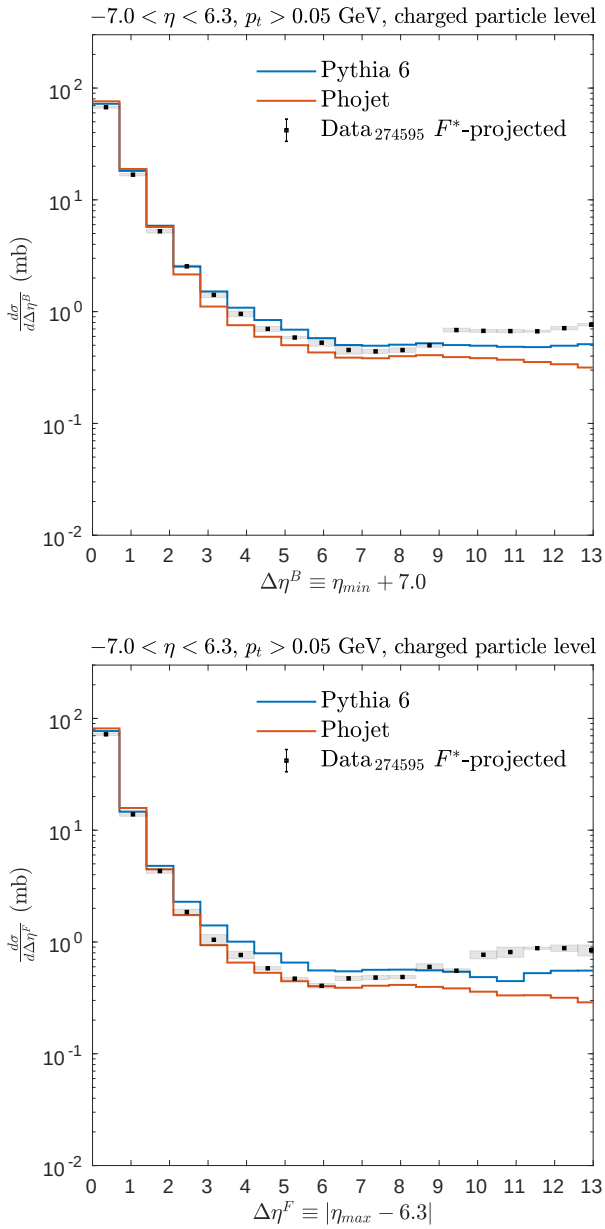

Figure 3.12: ALICE data at $\sqrt{s} = 13$ TeV and MC event generators: Type I pseudorapidity gap size distribution to the backward (top) and the forward (bottom) direction measured with respect to the detector edge. The projection uncertainties denote MC model variations.

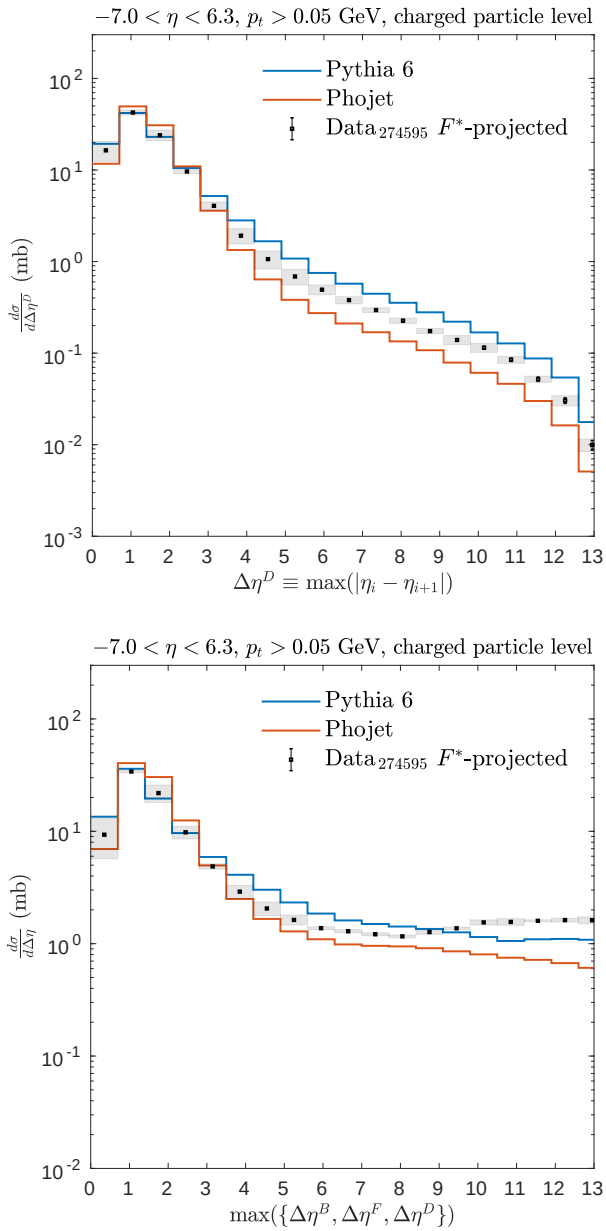

Figure 3.13: ALICE data at $\sqrt{s} = 13$ TeV and MC event generators: Type II pseudorapidity gap size distribution (top), and the maximum of type I and type II gaps per event (bottom). The projection uncertainties denote MC model variations.

One can see that the low mass diffraction, which dominates at large pseudorapidity gap sizes, is more prominent in data compared with Pythia 6 and Phojet event generators. The magnitude of low mass diffraction cannot be calculated from the Regge power law asymptotics controlled by the Regge trajectories, thus its proportion in event generators is purely model and tune dependent. At low masses, the proton structure fluctuations and baryonic resonances dominate. The values obtained here are in the line with the ATLAS measurement at $\sqrt{s} = 7$ TeV, which however has data only up to $\Delta\eta \approx 8$ [56]. They have an explicit control on the transverse energy cutoff, although the calorimeter noise starts to dominate over the signal at very low-p_t.

3.7 Conclusions

The main new results of the measurements presented in this section with the pp data at the cms energy $\sqrt{s} = 13$ TeV are as follows.

- ◇ The first unfolded multidimensional fiducial measurement of combinatorial partial cross sections.
- ◇ The first multidimensional maximum likelihood extraction of the single, double and non-diffractive cross sections. Re-extraction is possible a posteriori with future Monte Carlo models based on the measured unfolded partial cross sections.
- ◇ The first multidimensional extraction of the effective soft Pomeron intercept with $\Delta_P = 0.094 \pm 0.01$ (stat+syst).
- ◇ Low mass diffraction enhancement with respect to Monte Carlo event generator tunes is clearly visible in the forward pseudorapidity gap distributions up to $\Delta\eta \simeq 13$ units, projected using the newly developed F^*-algorithm.
- ◇ Our new measurements can be used to get more understanding of the AGK rule type calculus, which is only partially understood theoretically. These are used implicitly in the construction of the EPOS LHC and QGSJet-II-04 event generators, which give one of the best descriptions of the measured partial cross sections. However, the quality of MC parameter 'tunes' plays always a big factor. The detector level distributions in Appendix A.1 are also a step forward on this topic. A future measurement using $\mathbb{F}_q^N$ with $q = 3$ or more, with three different multiplicity bins per rapidity slice, could make this topic more transparent. This higher order measurement requires more statistics at least by a factor of $3^N/2^N$.

4 ALICE and *glueballs*

We introduce new algorithms for the studies of exclusive and semi-exclusive central diffraction (CED/CEP) and describe our measurements with ALICE data. Potentially the most interesting on this topic is the future identification of glueball resonances, understanding the pomeron spin structure and the contribution of proton dissociative events. Glueballs, the resonant bound states of non-abelian gauge fields, are very difficult objects because they mix quantum mechanically with quark bound states. A curious mathematical fact was pointed out by Coleman already in 1977, that *classical* non-abelian gauge fields do *not* contain glueball like field configurations [57]. Some predictions for the quantum glueballs can be calculated using lattice QCD [58, 59] for pure glue states within the 'quenched approximation', or using holographic approximations within Witten-Sakai-Sugimoto model based descriptions [60, 61]. Different mixing matrix scenarios have been considered also purely phenomenologically [62].

The experimental identification of glueballs is an extremely laborious task, requiring simultaneous measurements in multiply decay channels such as pion pairs, kaon pairs but also photon pairs. We know that gluons couple only to quarks or other gluons, so a (pure) glueball decay to photons should be suppressed via quark loops. Central exclusive production is in general considered to be the best glueball production channel at the LHC [63], pomerons being presumably objects with a high gluon content. Technically speaking all f-mesons are glueball candidates at some level, some more than others because of their position on the pomeron Regge trajectory such as $f_2(1950)$ or because of lattice QCD estimates, which is the case for example with $f_0(1710)$. This way we see that it is just natural to say

$$\text{Central Production Spectrum} \simeq \text{Glueball Candidate Spectrum}.$$

Also, we point out that there is no such strict spin-parity rule as $J^{PC} = 0^{++}, 2^{++}, ...$ in central production per se, quite often erroneously claimed. It is just that the *decay channels* which are usually measured, provide no other options by conservation laws. One may well produce resonance states with other quantum numbers too such as

axial vectors $J^P = 1^+$, as shown by the WA102 data [64]. In any case, the non-perturbative production couplings are currently theoretically incalculable.

The first evidence for the existence of double pomeron exchange type processes $pp \to p + X + p$ was measured at the CERN ISR in the low-energy range $30 \leq \sqrt{s} \leq 62$ GeV [65]. For a review of the pre-LHC era measurements see [66] and for recent measurements see Table 5.1. Our measurements do not include the forward protons but we work rather with the Babinet's complement: a forward veto type event selection. The event signature is a striking one for a high energy hadron collider: we select events with two reconstructed charge opposite tracks and require otherwise an empty detector. This semi-exclusive event selection will include also events with one or both of the forward protons excited into a low-mass state.

4.1 DrEM-PID: Double Recursive Expectation Maximization Particle Identification

For the particle identification, we developed a new fast maximum likelihood algorithm that we dubbed DrEM-PID. It is very simple and thus, similar to the previous algorithms such as described in [67]. However, ours is based on a double recursive (iterative) utilization of the classic Expectation Maximization (EM) algorithm, first for single particle tracks and then for particle combinations such as final state pairs. The obtained probabilities per event can be used as weights or as cuts with a suitable type I and type II frequentist error working point, also known as the ROC (receiver operating characteristics) working point. The weighted approach is the most streamlined regarding the PID related efficiency-purity corrections, in principle, no misclassification corrections are needed as long as all relevant particle pair combinations are included in the scheme, the fiducial momentum range is limited enough and the detector likelihood functions are high precision enough. In practise, some residual corrections may be necessary. Different strategies should be cross-compared using pure Monte Carlo input. We shall again remark that the EM-algorithm uses the Bayes formula, but is not a Bayesian method.

Phase I: In the first phase, one calculates the single track probabilities by EM-iteration as a function of the track 3-momentum $\vec{p}$, typically results binned over $|\vec{p}|$, for each three hadronic species π, K, p mass hypothesis. The detector information needs to be described in terms of signal likelihood functions, in the first approximation independently with

$$L_{tot}(\mathbf{x}, \vec{p}|h) = \prod_{i \in \text{detectors}} L_i(\mathbf{x}, \vec{p}|h), \tag{4.1}$$

where $h \in \{\pi, K, p\}$ and the product is over different detectors, such as the ionization dE/dx likelihood from the tracking detectors and the time-of-flight likelihood. If one of the detectors provides no signal, it is simply described with unit likelihood. These likelihoods are based on detector simulations for each particle hypothesis and the $\mathbf{x}$ denotes the set of low-level signal measurements. The ionization obeys a stochastic process with Landau distribution, but a sum of multiple ionization signals obeys Gaussian approximations by central limit theorem. The algorithm can handle different likelihood functions, also multidimensional beyond the product likelihood. The EM iteration obtains the particle fractions with

$$P(h|\mathbf{x}, \vec{p}) \leftarrow \frac{L_{tot}(\mathbf{x}, \vec{p}|h) P(h|\mathbf{x}, \vec{p})}{\sum_k L_{tot}(\mathbf{x}, \vec{p}|k) P(k|\mathbf{x}, \vec{p})}, \tag{4.2}$$

which is iteratively updated starting with uniform particle fractions until convergence. This is repeated independent for each kinematic hyperbin depending on $\vec{p}$.

Phase II: In the second phase for the charge identified and $\pm$ ordered tracks, one tabulates all 3^N hadronic combinations and uses the EM-iteration to extract the relative abundances of these by iterating

$$P(\{h\}_c) \leftarrow \frac{\prod_{i=1}^{N} P(h_c^{(i)}|\mathbf{x}_i, \vec{p}_i) P(\{h\}_c)}{\sum_{k\in\text{combs}} \prod_{i=1}^{N} P(h_k^{(i)}|\mathbf{x}_i, \vec{p}_i) P(\{h\}_k)}, \tag{4.3}$$

which yields the event-by-event probabilities for each of the final states.

A concrete example with $3^2 = 9$ combinations for $+-$ ordered pairs yields

$$\left\{\{\pi^+\pi^-\}, \{K^+K^-\}, \{p\bar{p}\}, \{\pi^+K^-\}, \{\pi^+\bar{p}\}, \{K^+\pi^-\}, \{K^+\bar{p}\}, \{p\pi^-\}, \{pK^-\}\right\}. \tag{4.4}$$

Only the first three are physical for fully exclusive central production, unless strangeness or baryon number is violated. However, in a semi-exclusive process all are easily possible. The relative abundances of these channels yields also a handle on the exclusivity of the data sample. Four particles gives already $3^4 = 81$ combinations, which is no problem when the combinations are generated algorithmically. When the track has large $|\vec{p}|$, the particle identification based on dE/dx will fail with any algorithm, only a good resolution time of flight measurement or for example Cherenkov radiation based measurement can then extend the identification momentum range. It is $|\vec{p}| \approx 0.6$ GeV when $\langle dE/dX \rangle$ of the TPC does not yield anymore strong discrimination between pions and kaons. For protons, this goes up to $|\vec{p}| \approx 0.9$ GeV. When the TOF signal is reconstructed, the kaon discrimination will extend approximately up to 2 GeV and for protons up to 4 GeV.

Basics of binary decision theory

We shall here point out a couple of crucial things about type I and type II errors, not always explained in a concrete manner. Let us first define these frequentist errors in Table 4.1. The *power* of the statistical test is given by $1-\beta$ whereas the statistical *significance* level is set by α.

Let us measure an observable x. The frequentist framework works purely with the likelihood functions $\mathcal{L}(x; H_i)$ of the signal and background and the optimal frequentist decision is given by the likelihood ratio via the Neyman-Pearson lemma [68]

$$R(x) = \frac{\mathcal{L}(x; H_0)}{\mathcal{L}(x; H_1)}, \quad \text{s.t.} \quad \alpha = P(R(x) \leq c), \tag{4.5}$$

	Background H_0	Signal H_1
Accept	Type I error: $P = \alpha$ (false pos.)	$P = 1 - \beta$ (true pos.)
Reject	$P = 1 - \alpha$ (true neg.)	Type II error: $P = \beta$ (false neg.)

Table 4.1: Frequentist decision theory probabilities P.

which yields the most powerful test with maximum true positive probability $1 - \beta$, that one can construct at the fixed significance level α. Algorithm: First one chooses the significance level of interest such as $\alpha = 0.05$, then one finds out the cut parameter c such that significance level holds. This can be done numerically with simulated likelihoods. The resulting decision domain can be very complicated already in one dimension $x \in \mathbb{R}$, if the likelihoods are complicated functions. The likelihood functions can be analytic or estimated by neural techniques, for example.

The likelihood ratio is a binary decision optimality with no prior information. The Bayes factor ratio of posteriori probabilities is the corresponding Bayesian version

$$\frac{P(H_0|x)}{P(H_1|x)} = \frac{\mathcal{L}(x;H_0)}{\mathcal{L}(x;H_1)} \frac{P(H_0)}{P(H_1)}. \tag{4.6}$$

By using the Maximum a Posteriori (MAP) decision, one reaches the so-called Bayes error which is the irreducible probability bound of misclassifications. A fully Bayesian approach will integrate over prior distributions, usually done with Monte Carlo sampling. One must understand that the PID classification efficiency and PID background (misclassification) corrections are different and have a different type of systematics with the frequentist and Bayesian philosophies. Then, in contrast, our fully weighted approach (which can be either frequentist or Bayesian) does not strictly speaking belong into the framework of hard binary decision theory but should be considered as a complete regression (fit) algorithm.

4.2 F2X: A Faster Analysis of Cross Sections

For the central production, but not limited to, we designed a new radically simple analysis chain driven by our algorithms. This minimal approach tries to mitigate the ever increasing abstraction and complexity of high energy physics analyses. The chain is as follows.

F2X in a nutshell

1. Tracking, fiducial cuts and vetoes
2. DrEM-PID final state identification weights
3. DeepEfficiency efficiency inversion weights
4. Luminosity weight
5. Data(!) events in HepMC3 containers
6. Event by event weighted analysis using only fiducial final state information

The idea is very simple: all experimental procedures are done event by event with probabilities. Final state identifications are executed with the DrEM-PID algorithm and efficiency corrections with the DeepEfficiency algorithm of Chapter 8. Finally, events together with the weights are stored in HepMC3 containers which allow to treat the data and Monte Carlo events through the same analysis and plotting machinery. In practice one may work with weighted histograms of observables. Also, only final state information is used so any internal Monte Carlo information about intermediate states or propagators, parton showers etc. is explicitly forbidden within this philosophy. To keep the measurement as a measurement. This philosophy is the same as with Rivet, but we go beyond simple histogram containers.

In principle, also negative events weights could be utilized e.g. for background corrections. Corrections that cannot be implemented with event-by-event weights such as background corrections can be done a posteriori with histogram subtractions, as usual. This type of corrections may be 'feed-down' of higher multiplicity final states to lower ones, for example, by detector efficiency losses. One must be careful in classifying the algebraic type of corrections either into a multiplicative or additive class. This type of analysis also works as a repository container of the data and its corrections.

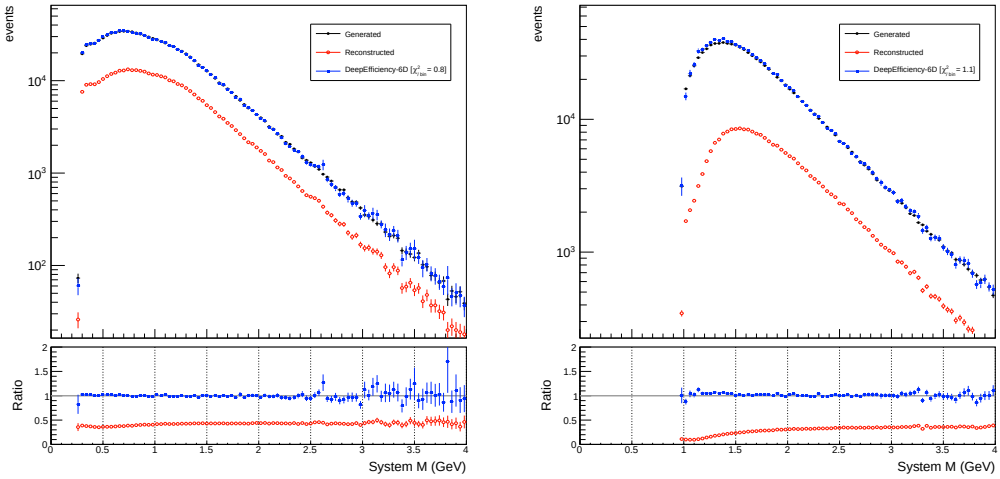

Figure 4.1: ALICE simulation: DEEPEFFICIENCY-6D based efficiency inversion of $\pi^+\pi^-$ and K^+K^- pairs. The reconstructed 1D-observable is the pair invariant mass.

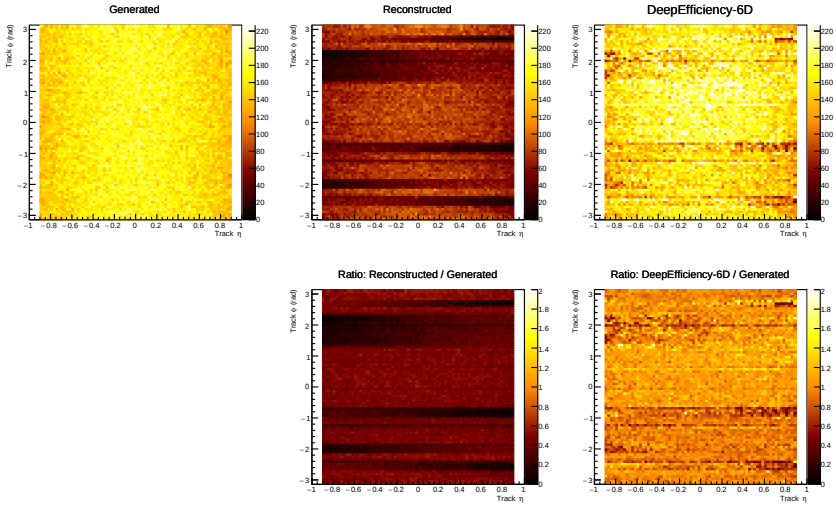

Figure 4.2: ALICE simulation: DEEPEFFICIENCY-6D based efficiency inversion of K^+K^- pairs. The reconstructed 2D-observable is the positively charged track (η, ϕ).

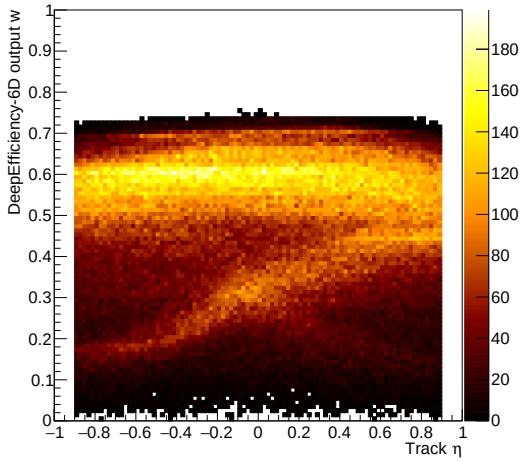

Figure 4.3: ALICE simulation: DeepEfficiency-6D event-by-event weight distribution projected for a single track pseudorapidity of the track pair.

We illustrate the DeepEfficiency algorithm efficiency inversion in Figures 4.1, 4.2 and 4.3 trained with approximately 5 million full GEANT simulated events. The 6D-input is the 3-momentum of two tracks which contains thus all correlations. The test samples are different than the training samples both statistically and systematically, by using slightly different MC generator level distributions. The inversion yields reliable results and the algorithm learns the very sharp discontinuities caused by the large inactive regimes in the SPD layers already with the modest MC training sample sizes. In these simulations we use the fiducial domain definition of $|\eta| < 0.9$ and $p_t > 0.15$ GeV, which yields nearly flat performance for the pion pairs and slightly worse for the low-mass kaon pairs. We estimate that with larger samples and the network architecture design optimizations the results will reach asymptotically near perfect inversion. However, in practice, one needs to study the performance through extensive simulations for the bias and variance. DeepEfficiency provides simultaneously a parametrization for fast detector efficiency simulations beyond the inversion application. It also provides efficiency corrections *simultaneously in all* observables. By using the EM-iteration, we could extend the algorithm to high dimensional unfolding context. However, a direct event-to-event inversion is not possible, because the unfolding mapping is not one-to-one. Thus, one needs to Monte Carlo sample densities, which poses new technical challenges due to new high dimensional density estimation techniques needed such as 'normalizing flows' [69].

4.3 Experimental tracking and PID setup

Tracking is based on the inner tracking silicon (ITS) detector, described earlier and the time projection chamber (TPC) [70] with a geometric acceptance of $|\eta| < 0.9$. The TPC is presumably the largest ionization gas chamber in the world currently with multi-wire proportional chambers (MWPC) end plate read out providing the cathode planes with 560k channels, which are in run III replaced with faster gas electron multipliers (GEM). The original run I gas mixture was 90% Ne and 10% CO_2 with a volume of 88 m^3, with some gas mixture modifications done at the run II. The gas is under a 400 V/cm electric field, yielding a drift time of 88 ms, which clearly limits any high collision rate use of the TPC. The electronic charge read out is coupled through low input impedance charge sensitive operating amplifiers and analog pulse shapers (PASA) and then sampled by 10 bit analog-to-digital converters at 5 to 12 MHz sampling frequency. The detector gas volume is calibrated through a built-in UV laser system, to compensate for any non-uniformity in the drift field and electron paths. These corrections are a standard part of the AliROOT event reconstruction chain executed in the computing grid.

Cut	Criterion
Reconstruction	ITS + TPC
TPC cluster per track	> 70
Track χ^2/ndf	< 4
DCA_T	$< 7\sigma$
DCA_z	< 0.5 cm
Vertex position	$\lvert z\rvert < 10$ cm

Table 4.2: Tracking low-level cuts, where DCA means 'distance of closest approach'.

The simultaneous track recognition and fitting algorithm are based on the Kalman filter, which is a recursive linear estimator for sequential measurements. It is an optimal algorithm in the case when the noise is Gaussian and the system is linear. Both are not exactly the case with tracking, however, it is still one of the best algorithms given its relative simplicity and well understood behavior. The alignment of the tracking is based on both cosmic rays and proton-proton collisions using different global and local fitting approaches, for more information see [71]. The warm solenoid magnetic field was operating under its default value of $B = 0.5$ Tesla. With ITS and TPC in use, the mass and transverse momentum resolution in the low mass 2-track measurements was simulated to be better than 0.5%. No unfolding of the

measurement is necessary with such a good resolution. We summarize the tracking cut parameters in Table 4.2.

ALICE has by construction numerous particle identification capabilities. The detectors which provided PID information in this study were the ITS ionization dE/dx and the TPC higher resolution dE/dx and the time-of-flight (TOF) detector for the track velocity $\beta = v/c$ measurements [72]. The TOF is built using multigap resistive plate chamber technology (MRPCs) with a total number of read out channels being 157k. The MRPCs are built using a stack of glass plates working as dielectric resistive material layers with conductive pickup cathodes on top and bottom of the stack, the anode plane set in the middle of the glassburger. In total there are 5 gaps of gas with the width of a gap being 250 μm, on top and bottom of the central anode. The incoming charged particle ionizes the gas in the stack and the charge is amplified using the high electric field resulting in an avalanche. The electrons are not free flowing through highly resistive glass layers, instead, the signal coupling is capacitive. The idea behind multiple layers is to be able to use a very high electric field of 100 kV/cm, resulting in a high signal efficiency together with a good time resolution.

The TOF time resolution in-situ is approximately 80 ps under optimal event conditions with intrinsic resolution of the MRPC elements measured to be approximately 50 ps. The TOF signal 'matching flow' efficiency with the ITS + TPC tracks starts to operate at $p_t \approx 0.3$ GeV, reaching approximately 50% efficiency at 0.5 GeV and the efficiency reaches plateau at 60 - 65 % around 1 GeV. To point out, this weak efficiency at low-p_t is naturally handled by probabilistic methods such as our DrEM-PID algorithm. When no TOF information is available, the product likelihood simply replaces the TOF term with unity and the overall tracking efficiency is left intact. The TOF measurement requires the interaction start time t_0, which is usually measured by the dedicated T0 detector of ALICE but in the case of low multiplicity events, it must be provided usually by the TOF itself yielding approximately 10 ps additional smearing on the time resolution compared to the T0 driven case. The third option is to use the LHC clock timing, providing t_0 with the resolution fundamentally limited by the LHC bunch spreading over the z-axis as $\propto \sigma_z/c$. It is an easy exercise to calculate how the (Gaussian) time resolution gives bounds to distinguish between particles of different mass as a function of the momentum $|\vec{p}|$. In the analysis, GEANT simulated signal templates for different particle hypothesis were used, used internally by the AliROOT PID libraries.

4.4 Semi-exclusive event selection

The event sample collected at the cms energy of $\sqrt{s} = 7$ TeV is based on a minimum bias trigger with VZERO + SPD and offline 'double rapidity gap veto'. Also, a much larger sample at the cms energy of $\sqrt{s} = 13$ TeV was collected with a special double rapidity gap trigger with a track candidate trigger requirements in the SPD and TOF detectors. These trigger requirements effectively give a much reduced tracking efficiency at low-p_t with complicated trigger systematics to be included in the future detector simulations, not available at this stage. Thus, we study here only the $\sqrt{s} = 7$ TeV sample with readily excellent efficiency, which is essentially flat over the system mass and transverse momentum in the fiducial domain given in Table 4.3.

Cut	Criterion
Track fiducial	2 tracks with $\|\eta\| < 0.9$, $p_t > 0.15$ GeV and charge $\sum_i Q_i = 0$
Exclusivity veto	Veto in $\eta \in [-3.7, -0.9]$ and $\eta \in [0.9, 5.1]$ (VZERO & FMD)
	No unassociated tracklets in SPD within $\eta \in [-2.0, 2.0]$
Track PID	ITS + TPC + TOF
	MAP decision for both tracks using *pp*-minbias Bayesian priors

Table 4.3: Event selection cuts.

The low-level event selection was developed during the years by numerous people with codes available under github.com/alisw/AliPhysics/tree/master/PWGUD. The high-level analysis codes of this work are available on my github. In this study, we require Maximum a Posteriori probability PID decision for both tracks (MAP⊗2), using kinematics dependent but otherwise fixed *pp*-minimum bias priors for the particle fractions. With larger statistics one could try to EM-fit the fractions from data, as described earlier. We compared the results with the fully weighted PID, with nearly identical mass distributions shapes and event counts. This is an expected result given the good performance of the PID for low-momentum tracks which dominate our sample.

The purity of the sample was studied counting the number of events using the exclusive selection, but requiring the charge sum $Q = \pm 2$. This was measured to be $< 1.5\%$ of the full sample after veto and track cuts, indicating good tracking and veto efficiency within the central domain, but is not a fully conclusive statement. Based on this, we may gain an estimate of the 'feed-down' background of $Q = 0$ from high multiplicity central production. For example, a system X with four charged pions with two lost tracks due to inefficiency will yield four possible sum $Q = 0$

combinations and $Q = \pm 2$. The four $Q = 0$ combinations will, unfortunately, pass the selection. With larger statistics, one could subtract this using the $Q = \pm 2$ spectrum multiplied by two. One could also in a data-driven way try to mathematically invert the 'complete ladder' of feed-down, by using the measured 2,4,6,...-track events with different final states. This requires solving a large system of possibly non-linear equations with some coefficients needed from Monte Carlo.

Final state	MAP⊗2	(%)	Weighted	(%)
$\pi^+\pi^-$	32608	(90.234)	32352.8	(88.776)
K^+K^-	958	(2.651)	1003.6	(2.754)
$p\bar{p}$	119	(0.329)	119.4	(0.328)
π^+K^-	1027	(2.842)	1233.1	(3.384)
$\pi^+\bar{p}$	163	(0.451)	190.5	(0.523)
$K^+\pi^-$	1097	(3.036)	1336.4	(3.667)
$K^+\bar{p}$	14	(0.039)	19.9	(0.055)
$p\pi^-$	136	(0.376)	166.9	(0.458)
pK^-	15	(0.042)	20.3	(0.056)
Σ	36137		36443.0	

Table 4.4: ALICE data at $\sqrt{s} = 7$ TeV: Integrated final state event counts with two different PID selection strategies.

The total number of $Q = 0$ events are shown in Table 4.4. The number of seemingly strangeness conservation 'violating' pairs of π^+K^- or π^-K^+ is thus approximately the same as K^+K^- pairs. These lonely kaons may be for example due to efficiency losses (feed-down) or forward excitation, which can leak them from the forward system fragmentation. Also, these may come from random minimum bias processes generating large rapidity gaps, even if with exponentially suppressed probability. The states $K^*(892)$ and $K^*(1430)$ which decay dominantly into pion-kaon pairs, were observed in the pion-kaon samples. Interestingly, PDG states that the (glueball candidate) $f_2(1950)$ is seen to branch into $K^*(892)\bar{K}^*(892)$. These decay channels should be measured to construct a clear hierarchy of f-mesons and glueballs. We remark that we see some asymmetry in Table 4.4 at the level of a few standard deviations between the charge conjugate channels, possibly due to material interaction effects and a calibration mismatch. We conclude that extensive MC background studies are needed for the future cross section measurements, with complete efficiency and background corrections, which goes beyond what is shown here. We shall now concentrate on the pion and kaon pair channels spectroscopy.

4.5 System invariant mass spectrum

To study the invariant mass spectrum in detail we implemented a simple, but complete spectrum fit. We point out that theoretically better fits can be done with our Graniitti MC, where individual phenomenological scattering amplitudes are summed together at amplitude level and evaluated as a function of the Lorentz scalars. The corresponding fit code is part of the package. However, for simplicity we shall fit the spectrum this way. The total spectrum parametrization is

$$f(m) = \left| \sum_k A_k e^{i\phi_k} g(m; M_0^{(k)}, \Gamma_0^{(k)}) + \psi(m) \right|^2 , \tag{4.7}$$

where the sum is over resonances with complex weights $A_k e^{i\phi_k}$. We take the sum over four resonances in the case of $\pi^+\pi^-$ pairs and over two resonances in the case of K^+K^- pairs. For the underlying continuum, we take an ansatz

$$\psi(m) = iA \left[\frac{m-D}{\gamma} e^{-\frac{m-D}{\gamma}} \right]^{1/2} , \tag{4.8}$$

where A is the weight, γ is the scale and D is the shift parameter. The resonances are parametrized with relativistic Breit-Wigner poles

$$g(m; M_0, \Gamma_0) = \frac{\sqrt{2m}}{m^2 - M_0^2 + iM_0\Gamma(m)}, \tag{4.9}$$

where M_0 is the pole mass and $\Gamma(M)$ is the momentum dependent width [73]

$$\Gamma(m) = \Gamma_0 \frac{M_0}{m} \left[\frac{m^2 - 4m_f^2}{M_0^2 - 4m_f^2} \right]^{3/2} , \tag{4.10}$$

where m_f is the decay daughter mass. The denominator factor $\sqrt{2m}$ originates from the change of variables $|G(m^2)|^2 dm^2 \mapsto |g(m)|^2 dm$ of the absolute squared density of the relativistic propagator expression $G(m^2) = 1/[m^2 - M_0^2 + iM_0\Gamma]$. This needs to be taken into account, because our expressions are as a function of m.

We remark that the fit exercises with Breit-Wigners are complicated for composite (non-elementary) resonances with varying spins, phase space and production channel effects. No simple parametrization is expected to hold perfectly.

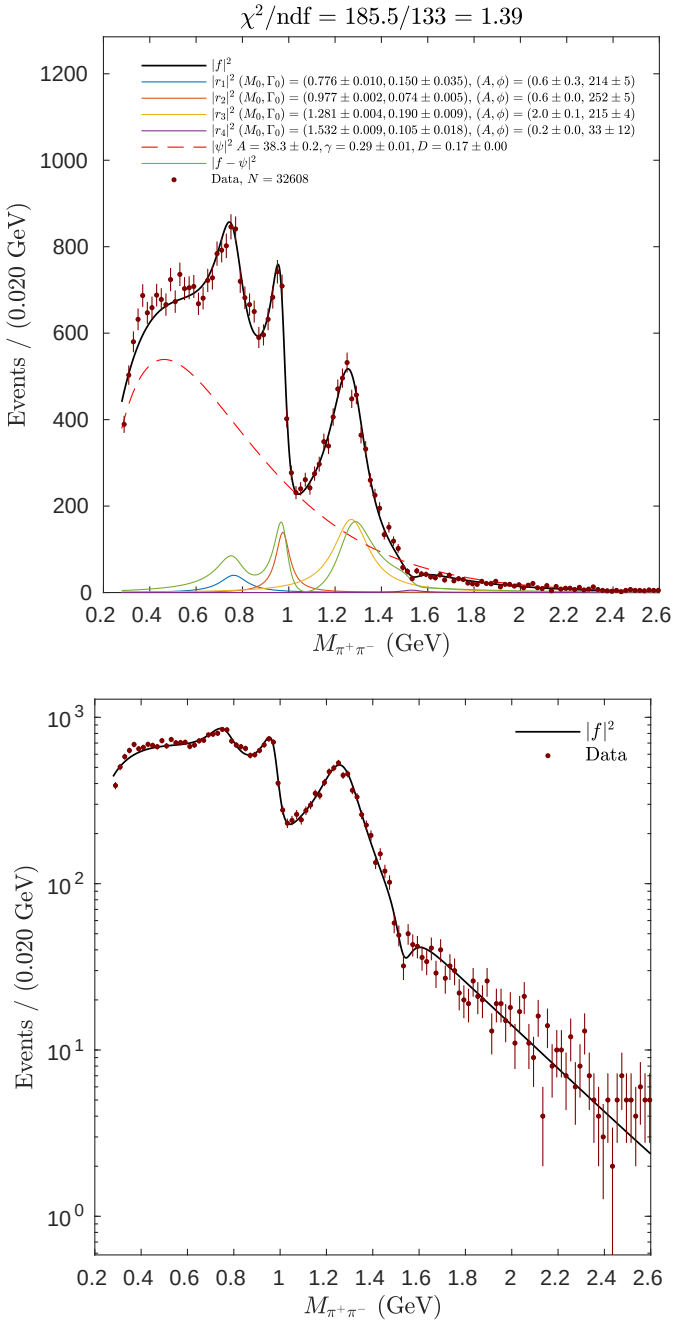

Figure 4.4: ALICE data at $\sqrt{s} = 7$ TeV: The semi-exclusive $\pi^+\pi^-$ spectrum with $|\eta(\pi)| < 0.9$ and $p_t(\pi) > 0.15$ GeV.

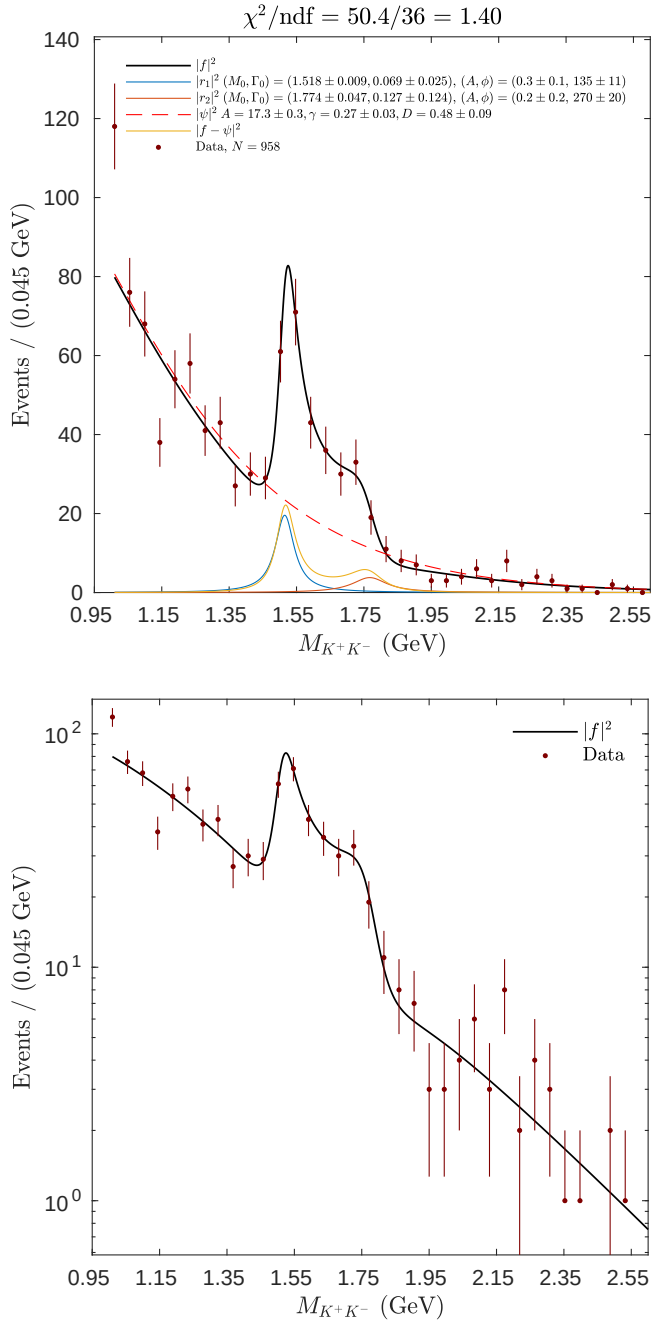

Figure 4.5: ALICE data at $\sqrt{s} = 7$ TeV: The semi-exclusive K^+K^- spectrum with $|\eta(K)| < 0.9$ and $p_t(K) > 0.15$ GeV.

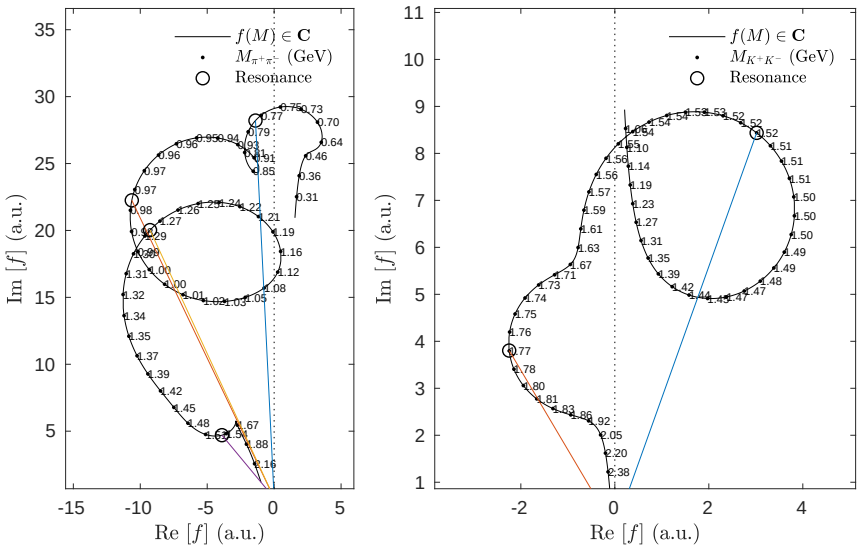

Figure 4.6: ALICE data at $\sqrt{s} = 7$ TeV: The complex plane fit curves in $\pi^+\pi^-$ on the left and in K^+K^- on the right. Colored lines are for visualizing the resonance pole positions.

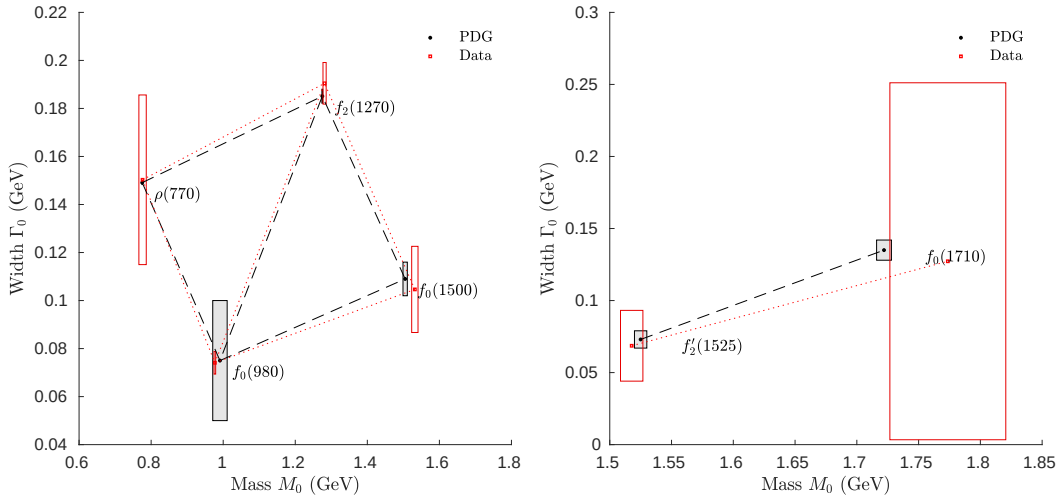

Figure 4.7: ALICE data at $\sqrt{s} = 7$ TeV: The fit extracted resonance parameters with statistical 1σ uncertainties and comparisons with the established PDG resonance candidates in $\pi^+\pi^-$ on the left and in K^+K^- on the right.

The results for the $\pi^+\pi^-$ and K^+K^- spectrum and the corresponding parameter values with statistical uncertainties (phases in degrees) are shown in Figures 4.4 and 4.5, and the extracted resonance mass and width parameters are compared with the closest matching PDG resonance values in Figure 4.7. The fits have near unity χ^2/ndf values. The amplitude for the $f_2(1270)$ candidate is significantly larger than for the other resonances, which can be also seen by fitting the spectra by using Graniitti scattering amplitudes, so it is not just an effect from this particular fit parametrization. This is interesting given that it is a spin-2 state and not a spin-0 (ground state). The obtained statistical fit uncertainties for the mass and width of the $f_0(980)$ candidate are very competitive, better than the PDG global average. We see a sharp drop in the invariant mass spectrum near $M_{\pi^+\pi^-} \simeq 1$ GeV, which may have something to do with the K^+K^- channel opening and unitarity. Fitting the mass spectrum around that point relies on complex interference. The total fit functions in the complex plane are shown in Figure 4.6.

We observe that the continuum slope γ has the same value for both of the final states within statistical uncertainties, yielding knowledge about the non-resonant scattering dynamics. This could be easily compared with Graniitti based simulations, by adjusting there the continuum amplitude off-shell form factor parametrizations. It can shed light on the question is the continuum amplitude dominated by a sub-t channel exchange or not. At very low masses, the enigmatic $f_0(500)$ resonance was not manifestly present given that the continuum parametrization alone was able to describe the data. Again, this is something to be studied in more detail with Monte Carlo and spherical harmonic angular decompositions. The identification of possible glueball states from the spectrum requires further systematic studies. Because glueballs are expected to mix with ordinary meson states, to decompose the mixing experimentally especially without forward protons, one needs do simultaneous high statistics measurements of pion, kaon and gamma pair final states. Also the four body decays should be measured, which can be used to probe resonances decaying to intermediate vectors such as $\rho^0(770)$ or $\phi(1020)$ pairs. Our DeepEfficiency algorithm solves the problem of unknown angular distributions affecting the efficiency corrections, which can be hard to implement otherwise with soft 4-body final states.

4.6 System transverse momentum

The system transverse momentum is a crucial observable, related to the transverse plane Gribov diffusion and the proton incoherent structure fluctuations. In this section we describe a fast Monte Carlo template fit to the system transverse momentum spectrum. As a stochastic distribution driving the fit templates, we use for the elastic coherent proton p_x, p_y-components the Gaussian distribution and for the excited incoherent proton p_t^2, we use the Gamma distribution ansatz. The forward-backward system transverse angle separation $\Delta\varphi$ is sampled from the uniform distribution, which is the Occam's razor choice without forward proton measurements. Then we get the central system transverse momentum by momentum conservation, and we also use the vertex factorization assumption for different legs in the process. The corresponding distribution parameters together with the process fit fractions f_i are shown in the legend of Figure 4.9, where b is the coherent forward proton exponential p_t^2-slope related the the Gaussian components by $b = 1/(2\sigma^2)$, as derived in Appendix B.3. The parameter k is the shape of the incoherent Gamma distribution with the scale parameter set as $\theta = 1/b$, and the mean is given by $k\theta$. Effectively, the Gamma distribution includes both exponential and power law (hard tail) behavior.

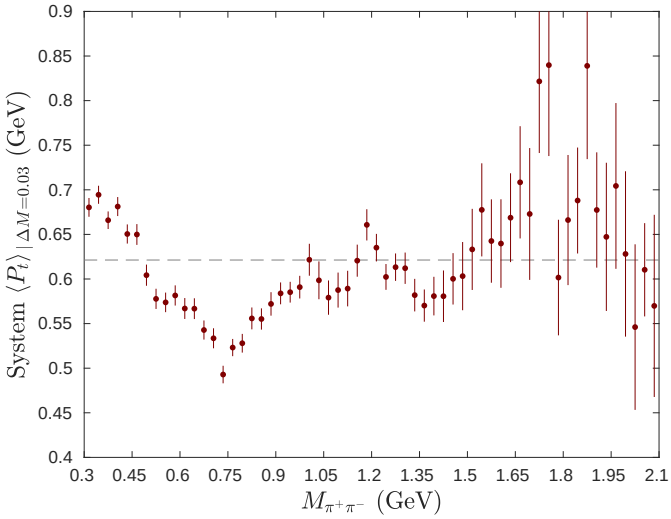

Figure 4.8: ALICE data at $\sqrt{s} = 7$ TeV: The $\pi^+\pi^-$ system mean transverse momentum as a function of the invariant mass with $|\eta(\pi)| < 0.9$ and $p_t(\pi) > 0.15$ GeV. The dashed line is the mean transverse momentum over the full mass range.

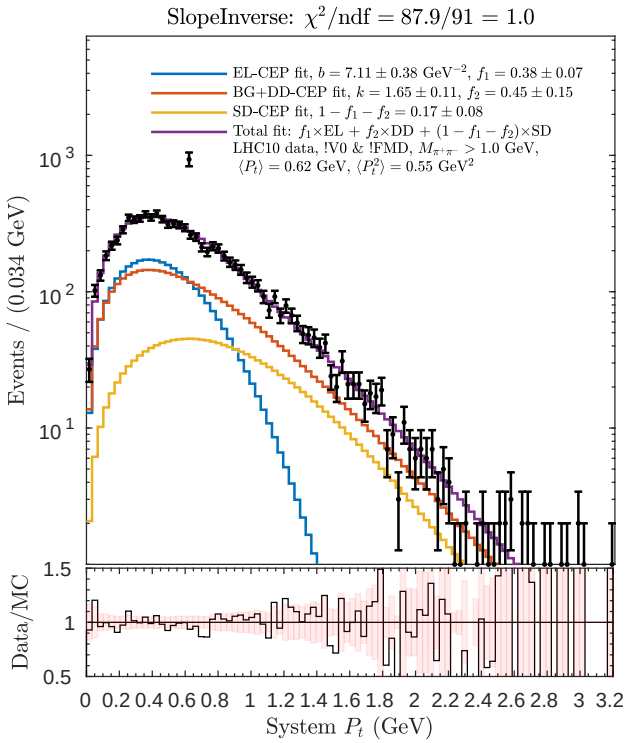

Figure 4.9: ALICE data at $\sqrt{s} = 7$ TeV: The $\pi^+\pi^-$ system transverse momentum fit with $M_{\pi^+\pi^-} > 1$ GeV, $|\eta(\pi)| < 0.9$ and $p_t(\pi) > 0.15$ GeV.

We use a minimum mass cutoff at 1 GeV, to remove the contribution from the photoproduction of ρ^0. The mean transverse momentum $\langle P_t \rangle$ as a function of the invariant mass is shown in Figure 4.8. We observe a clear dip a bit below the ρ^0 pole mass, which is a strong indication for the photoproduction origin. The figure shows also up and down variation near other resonances, which is expected. Now based on the fit shown in Figure 4.9, we obtain the fraction of events with elastic forward protons

$$f_{el} = 0.38 \pm 0.07\,(\text{stat}) \tag{4.11}$$

using the event selection of Table 4.3. The rest are inelastic events with one or two protons dissociated, which still pass through the forward veto. This ratio is compatible with Graniitti based simulations given in Chapter 5, which are based on a more sophisticated phenomenology, with MC parameters fixed by completely

different measurements. We found out that the ratio between single and double dissociation is strongly dependent on the model details, such as the vertex factorization and the incoherent stochastic distribution ansatz. It can be studied reliably only with larger event statistics. This fraction extraction is presumably done for the first time at the LHC, at least with the ALICE data. The fraction depends naturally on the forward veto acceptance of the experiment, thus a careful fiducial calibration should be done.

The obtained exponential elastic slope is

$$b = 7.11 \pm 0.38\,(\text{stat})\ \text{GeV}^{-2}. \tag{4.12}$$

The value is realistic and gives confidence to the elastic fraction estimate of the sample. Systematic errors in these are essentially a fit model driven, something which needs to be studied and simulated with full Graniitti based fits in the future, which allows one also to study e.g. the effects of screening corrections. For a comparison, the ATLAS+ALFA experiment obtained a value of $b = 7.65 \pm 0.34$ GeV^{-2} in single diffraction measurements at $\sqrt{s} = 8$ TeV [54].

4.7 Summary

The main results of the studies with semi-exclusive pp data with identified pion and kaon pair final states at the cms energy $\sqrt{s} = 7$ are as follows.

- ◇ The $\pi^+\pi^-$ continuum and the enigmatic $f_0(500)$ are ambiguous, that is, the simple spectrum fit cannot uniquely separate these. There is a sharp fit interference at $M_{\pi^+\pi^-} \simeq 1$ GeV, also visible by eye in the sharp drop of the invariant mass spectrum. Perhaps related to the K^+K^- channel opening and unitarity.
- ◇ The single dimensional complex spectrum fit is easily unstable in way that a heavy functional dependence was observed with the continuum parametrization ansatz. However, within statistical uncertainties, the same continuum parametrization slope γ was extracted for the pion and kaon channels, interesting for the phenomenology.
- ◇ The $f_0(980)$ mass and width extraction seems to yield significantly smaller (statistical) uncertainties than the PDG global average.
- ◇ The fit amplitude A_k of $f_2(1270)$ is a factor of a few larger than for the other resonances, a similar observation can be made with Graniitti MC based fits. This is somewhat puzzling.
- ◇ Photoproduction of $\rho(770)$ is visible, which was verified also qualitatively observing a dip in the system $\langle P_t \rangle$ distribution, also $\phi(1020)$ seems to be visible given the good low-p_t efficiency.
- ◇ Glueball candidates such as $f_0(1710)$ require more statistics to make systematic branching ratio comparisons feasible. Technically, all f-mesons are at least mixed candidates for glueballs.
- ◇ The visible fraction of events with elastic vs inelastic central production and the corresponding elastic proton b-slope in the process were extracted through fast Monte Carlo template fits, with values compatible with other measurements and phenomenology.

5 GRANIITTI: A Monte Carlo Event Generator for High Energy Diffraction

We describe the physics and computational power of GRANIITTI Monte Carlo event generator, a new fully multithreaded engine designed for high energy diffraction, written in modern C++. The emphasis is especially on the low-mass domain of central exclusive processes of the S-matrix, where exotic QCD phenomena such as glueballs are expected and holographic dualities with gravity may be tested. The generator includes photon-photon, photon-pomeron, pomeron-pomeron, Durham QCD model and Tensor pomeron model type scattering amplitudes and advanced spin analysis tools. The generator combines a full parametric resonance spectrum with continuum interference and forward + central spin correlations together with event-by-event eikonal screening loop and forward proton excitation kinematics. We demonstrate the state-of-the-art capabilities and show how the enigmatic 'glueball filter' observable is driven by spin polarization.

Chapter in: arXiv:1910.06300 [hep-ph]

5.1 Introduction

High energy diffraction probes 'asymptotic' strong interactions in the kinematic domain of small invariant momentum transfer $-t$ and large energy squared $s \to \infty$ or small x. The main processes of interest here are

$$pp \to p^{(*)} + X + p^{(*)}, \tag{5.1}$$

where the system X is produced via color and charge singlet exchanges between two protons, via photon or pomeron fusion. These enigmatic objects known as pomerons were constructed originally in the complex angular momentum theory of scattering amplitudes $\mathcal{A}(s, j)$, a method devised by Regge first in a non-relativistic scattering context [74] and then extended to the relativistic domain by Gribov, who also built the most intense but incomplete interacting pomeron calculus [75]. It is clear that one cannot currently consider these asymptotics as being properly explained by current QCD in the soft domain. But it is the future theory together with novel measurements which should eventually provide the detailed 'space-time picture' with fluctuating proton structure.

This 'Pomeranchuk' singularity can be a simple soft pole or a mathematically almost arbitrary complex set of cuts, such as described in the pioneering work by Mandelstam [76]. Later, a hard pomeron as a fixed branch point singularity was found in the field theory by Balitsky, Fadin, Kuraev and Lipatov [77] (BFKL). In general, this property of field theories is known as 'Reggeization' of the strongly interacting amplitudes in the Regge asymptotic. In gauge/string duality, the soft and hard pomeron have been unified [78]. Well known is that the Regge amplitudes behaving like $\propto s^{\alpha(t)}$ have a Gribov diffusion picture, obtained via a Fourier transform to the impact parameter space. They are directly related to the quantum mechanical stochastic branching random walk. An interesting discussion of the parton model, strings and black holes by Susskind is fundamental in this context [79], after all, we are talking about asymptotics. The elastic scattering asymptotics of the proton are fascinating in terms of the evolution of the 'blackness' of the disc and the transverse size of this disc, noting that we are now talking about average properties.

In our main process illustrated in Figure 5.1, the forward protons may stay coherent which is known as the central exclusive process or fluctuate into a dissociated state, denoted with p^*, giving semi-exclusive processes experimentally. The dissociation will result in a higher transverse momentum for the system X, experimentally one of the only signs of low mass dissociation bypassing the forward veto detectors. Other indirect observables may be related to the central system spin polarization and

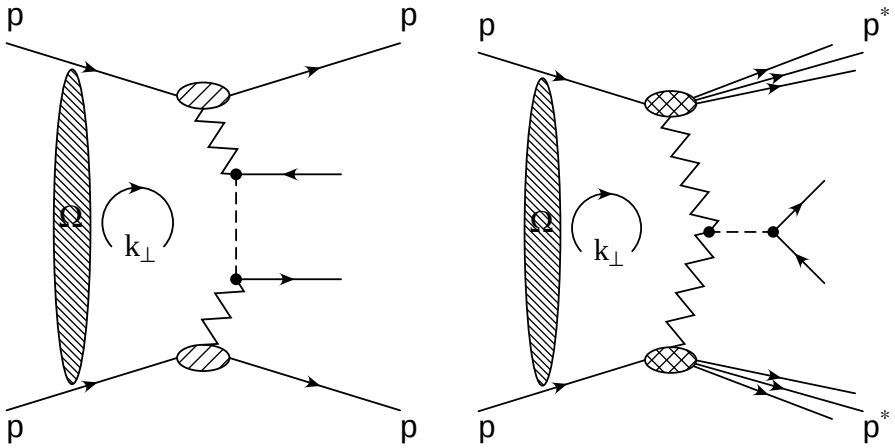

Figure 5.1: A pomeron-pomeron continuum amplitude on the left and a pomeron-pomeron resonance production process with eikonal Ω screening and double dissociation of the forward protons on the right.

relative magnitude of resonances. The emphasis here is in the non-perturbative domain of strong interactions, however, also certain perturbative QCD, QED and EW processes are included. The design of the engine is based on a synthesis $\leftrightarrow$ analysis philosophy, where the analysis side is accelerated via automated phenomenology of fiducial observables and spin polarization decomposition algorithms. The purpose of the Graniitti Monte Carlo event generator is to provide the necessary tools for the LHC experiments and beyond.

Some of the currently unique features of Graniitti, to our knowledge, are: central production of arbitrary spin $J = 0, 1, 2, 4, \ldots$ resonances and their spin polarization driven helicity amplitudes together with a non-resonant continuum, a full effective Lagrangian Tensor pomeron model implementation and forward proton dissociation combined with a screening (absorption) loop integral. The full potential of the generator is obtained when interfaced together with forward fragmentation such as generated by Pythia [80], especially interesting and useful in the case of ALICE and LHCb experiments without forward protons. Experiments with forward protons such as CMS+TOTEM, ATLAS+ALFA or STAR will be highly interesting due to the helicity driven 'glueball filter' type forward-central correlations. Thus both types of measurements will be crucial, their cross sections bootstrapping the theory together like a generalized Babinet's principle.

Outline This article is organized as follows. We first describe dynamics involved, then kinematics, analysis algorithms and briefly the technology behind the generator. Finally, we give some conclusions.

5.2 Dynamics

The scattering amplitudes currently included span from QED, EW, Regge theory and QCD. Advanced users can add more amplitudes e.g. from MadGraph, the normalization conventions between the phase space and matrix element squared obey the standard field theory conventions.

Eikonal pomeron

The soft pomeron is an enigmatic object in terms of QCD, however, it is the best effective description of diffraction or asymptotically slowly growing total cross sections so far. The simple eikonal pomeron model we use is based on the model from [81], however our eikonalization procedure is slightly different. The single pomeron exchange elastic pp-scattering amplitude is the basic building block

$$\mathcal{A}_{el}(s,t) = [g_{ppP}F(t)]^2 \eta(\alpha_P(t),+)\left(\frac{s}{s_0}\right)^{\alpha_P(t)}, \tag{5.2}$$

with the proton-proton-pomeron coupling g_{ppP} (GeV^{-1}) and the proton form factor parametrization

$$F(t) = \frac{1}{1-t/a_1}\frac{1}{1-t/a_2}, \tag{5.3}$$

with two free parameters a_1, a_2 (GeV^2). The coupling and form factor represent together a factorized real valued residue function at the vertex. The pomeron propagator is $(s/s_0)^{\alpha(t)}$ with the trajectory $\alpha_P(t)$ and the (even +) signature phase factor for the pomeron

$$\eta(\alpha(t),\pm) = -\frac{1 \pm \exp(-i\pi\alpha(t))}{\sin(\pi\alpha(t))}, \tag{5.4}$$

which comes from the Sommerfeld-Watson transform [82]. It is an analytical integral continuation of the integer spin partial wave series – the mathematical basis behind the Regge pole exchanges with $\alpha(t) \sim J$. This expression has trivial singularities (poles) at integer values of $\pi\alpha(t)$, thus $t \to 0$ limit is sometimes taken. The elastic amplitude gives us, by the optical theorem, the total cross section behavior $\sigma_{tot} \sim s^{\alpha_P(t)-1}$. Experimentally at high energies, the intercept $\alpha_P(0)$ needs to be larger than 1. We use a non-linear parametrization of the pomeron trajectory with a pion loop, described in detail in [81].

We note here that it is the signature factor which gives the ratio between real and imaginary part of the amplitude. In order to proceed with the eikonal model, the

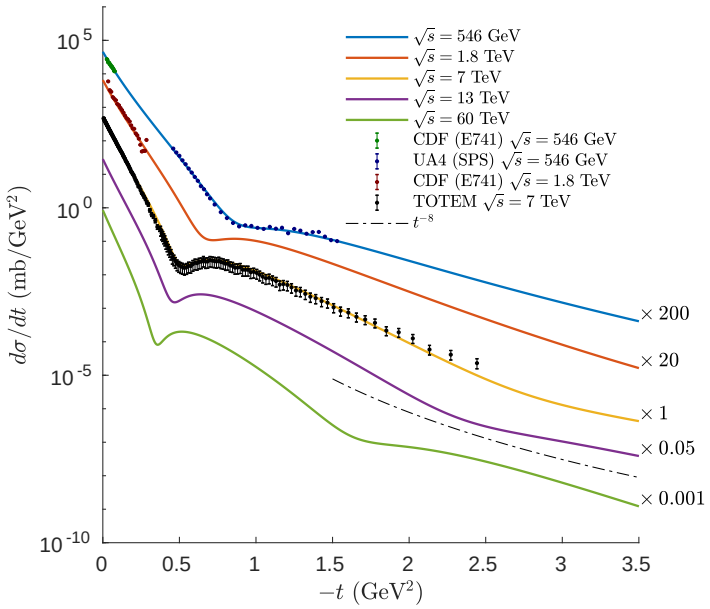

Figure 5.2: Elastic proton-proton scattering with the eikonal pomeron. Data from [83–85].

complex density in the impact parameter b_t-space is obtained by numerical (inverse) Fourier 2D-transform with proper normalization

$$\Omega(s, b_t) = \frac{1}{(2\pi)^2 s} \int \mathcal{A}_{el}(s, -\mathbf{k}_t^2) e^{i\mathbf{b}_t \cdot \mathbf{k}_t} d^2\mathbf{k}_t \tag{5.5}$$

$$= \frac{1}{2\pi s} \int_0^\infty \mathcal{A}_{el}(s, -\mathbf{k}_t^2) J_0(bk_t) k_t \, dk_t, \tag{5.6}$$

where J_0 is the 0-th Bessel function of the first kind – that is, we used the Fourier-Bessel transform because there is no azimuthal dependence. Now under the eikonal approximation, the multiple re-scattering amplitude which is an approximation to the more complicated cut singularities than just a single pomeron pole exchange, is given by exponentiation

$$\mathcal{A}_{el}^{eik}(s, b_t) = i \left(1 - \exp\left(i\Omega(s, b_t)/2\right)\right). \tag{5.7}$$

Then finally, we get the eikonalized amplitude back in momentum space with a

(forward) Fourier 2D-transform

$$\mathcal{A}_{el}^{eik}(s, t = -\mathbf{k}_t^2) = 2s \int \mathcal{A}_{el}^{eik}(s, b_t) e^{-i\mathbf{b}_t \cdot \mathbf{k}_t} d^2\mathbf{b}_t \tag{5.8}$$

$$= 4\pi s \int_0^\infty \mathcal{A}_{el}^{eik}(s, b_t) J_0(bk_t) b_t \, db_t. \tag{5.9}$$

We calculate these integrals automatically for each cms energy $\sqrt{s}$, generate look-up tables and interpolate the values event-by-event. The eikonalized amplitude is used to generate elastic scattering events and in the screening loop for the central production.

Now using unitarity (optical theorem), we get total, elastic and inelastic cross sections as

$$\sigma_{tot}(s) = 2 \int \mathrm{Im}\mathcal{A}_{el}^{eik}(s, b_t) \, d^2\mathbf{b}_t \tag{5.10}$$

$$\sigma_{el}(s) = \int |\mathcal{A}_{el}^{eik}(s, b_t)|^2 \, d^2\mathbf{b}_t \tag{5.11}$$

$$\sigma_{inel}(s) = \int 2\mathrm{Im}\mathcal{A}_{el}^{eik}(s, b_t) - |\mathcal{A}_{el}^{eik}(s, b_t)|^2 \, d^2\mathbf{b}_t. \tag{5.12}$$

The integrals are taken numerically using $\int \ldots d^2\mathbf{b}_t \to 2\pi \int \ldots b_t \, db_t$. We point out here that the 'optimal' pomeron trajectory parameters are significantly different within the eikonalized amplitude and non-eikonalized single exchange. This means that one simply cannot take the eikonalized pomeron and use that naively with central production amplitudes, for which we use the effective linear pomeron $\alpha_P(t) \simeq 1.08 + 0.25t$.

The characteristic shrinking of the 'diffractive cone' is shown in Figure 5.2, which comes out naturally and was originally predicted by Gribov. The dip structure of data is reproduced. We remark here that at high $-t$, a secondary dip develops due to the eikonalization procedure, which is a well known feature with a single Pomeron amplitude. Depending on the non-linear pomeron trajectory parameters, the proton form factor and the fit dataset extension to higher values in $-t$, it can be tuned to flatten out at LHC energies and to appear only at larger energies. Thus using it as a test statistic or model separation tool is not completely conclusive. From perturbative QCD at large values of $-t$ one expects a power law t^{-8} dependence. This amplitude is driven by 3-gluon exchange mechanism, with gluons between three different quarks [86, 87], which is a $C = -1$ exchange. See [87] for discussions about the expected dip behavior in pp and $p\bar{p}$.

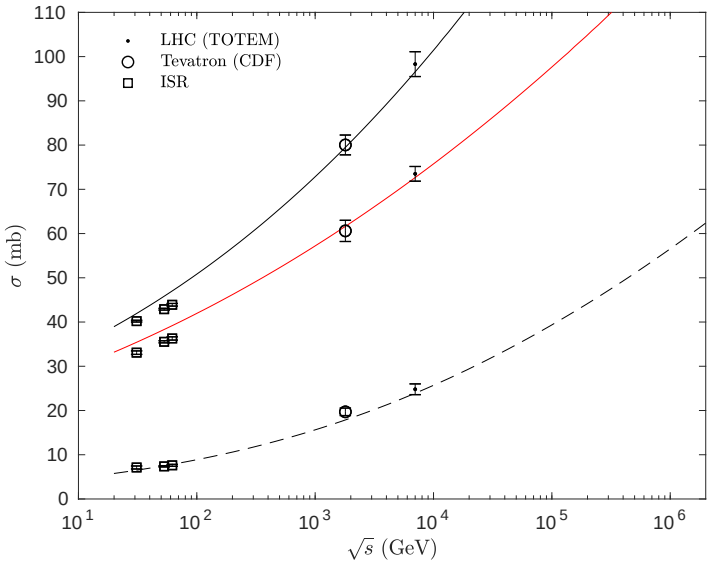

Figure 5.3: The eikonal pomeron based calculations of the total (black), inelastic (red) and elastic (dashed) proton-proton cross sections. Data from [88].

The behavior of total, inelastic and elastic cross section is shown in Figure 5.3. We see that at ISR energies the inelastic cross section, and thus the total, are slightly higher than what is being calculated after parameter tuning, indicating that additional degrees of freedom are important. This is well known phenomenologically, given that Reggeon trajectories have $\alpha_R(t) \simeq 0.5 + 0.9t$, a vanishing role asymptotically due to intercept $\alpha_R(0) < 1$, but a significant at lower energies. Our engine includes automated fit machinery to improve, simplify or extend the eikonal pomeron description and fit more extensive datasets.

Unitarity via screening loop

The screening loop can be turned on for all processes. The basic idea is that the screening loop will induce 'cut contributions', via the eikonal pomeron, to the central production which makes the total scattering amplitude unitary and in effect significantly suppresses the cross section. The screening integral is calculated event-by-event with 4-momentum conservation along vertices. The bare amplitude + loop

amplitude give the total screened amplitude

$$\mathcal{A}^S(s, \mathbf{p}_{t,1}, \mathbf{p}_{t,2}) \equiv \mathcal{A}^0(s, \mathbf{p}_{t,1}, \mathbf{p}_{t,2}) + \mathcal{A}^{loop}(s, \mathbf{p}_{t,1}, \mathbf{p}_{t,2}), \tag{5.13}$$

where for notation purposes, we leave out the dependence on the other kinematic variables, only implicitly affected by the loop integral. We do reconstruct the full kinematics for each discrete loop evaluation point.

As a high energy limit, the loop integral is done numerically in the 2D-transverse momentum $\mathbf{k}_t$-space

$$\mathcal{A}^{loop}(s, \mathbf{p}_{t,1}, \mathbf{p}_{t,2}) = \frac{i}{s} \int \frac{d^2\mathbf{k}_t}{8\pi^2} \mathcal{A}^{eik}_{el}(s, -\mathbf{k}_t^2) \mathcal{A}^0(s, \mathbf{p}_{t,1}, \mathbf{p}_{t,2}), \tag{5.14}$$

where $-\mathbf{p}_{t,1} + \mathbf{k}_t = \mathbf{q}_{t,1}$, $-\mathbf{p}_{t,2} - \mathbf{k}_t = \mathbf{q}_{t,2}$ with $\mathbf{q}_{t,1,2}$ being the transverse momentum transferred to the central system and $\mathbf{p}_{t,1,2}$ the transverse momentum of forward systems (protons). The screening has the largest differential impact to the forward observables, as illustrated in Figure 5.4.

Now relatively straightforward but by no means a complete generalization here would be to consider a 'multichannel' eikonal amplitude, responsible for the proton structure and its excitations. That is, the incoming proton is treated as a coherent superposition of eigenstates $|p\rangle$, $|p^*\rangle$, $|p^{**}\rangle \ldots$ and the eikonal screening is calculated over the set of pair of states and the corresponding eikonal densities. This approach for the screening loop is in use in Dime [89] and SuperChic [90] event generators. This shall be investigated in the future with more precise data to fit the parameters. Technically, our code includes already some prototype constructions. Perhaps, one could use new approaches for describing the Good-Walker excitations, such as lattice Yang-Mills simulation driven [91], a bit like its application in photoproduction in [92]. A classic approach that does not consider details of the structure is described in [81]. Our minimal description philosophy requires that each new free parameter for the dissociation part should be well motivated by data.

In Table 5.1 we show how the screening loop suppresses the cross sections for processes with a different type of scattering amplitudes and compare with experimental measurements. Clearly, photon-photon processes producing lepton pairs l^+l^- or W^+W^- pairs are more peripheral in $\mathbf{b}_t$-space with smaller average transverse momentum whereas pomeron-pomeron production of pion pairs results in larger average absorption

$$\langle S^2 \rangle \equiv \frac{\int d\Pi_N \left| \mathcal{A}^0(\Pi_N) + \mathcal{A}^{loop}(\Pi_N) \right|^2}{\int d\Pi_N |\mathcal{A}^0(\Pi_N)|^2}, \tag{5.15}$$

which is the phase space integrated cross section ratio between screened and unscreened processes. We denote the full abstract set of 4-momentum variables by Π_N

			measurement	graniitti		
	$\sqrt{s}$	phase space cuts	value $\pm$ stat $\pm$ syst	σ_S	σ_0	$\langle S^2 \rangle$
gg	13	$\lvert y\rvert < 2.5, p_t > 20$ GeV		0.872	14.1 nb	0.06
$\pi^+\pi^-_{EL}$	7	$\lvert\eta\rvert < 0.9, p_t > 0.15$ GeV		3.84	19.6 μb	0.20
$\pi^+\pi^-_{SD}$	7	$\lvert\eta\rvert < 0.9, p_t > 0.15, M_1 < 5$ GeV		1.25	9.34 μb	0.13
$\pi^+\pi^-_{DD}$	7	$\lvert\eta\rvert < 0.9, p_t > 0.15, M_{1,2} < 5$ GeV		0.269	3.13 μb	0.09
$\pi^+\pi^-$	7	$\lvert Y_X\rvert < 0.9$		9.27	45 μb	0.21
$\pi^+\pi^-$	0.2	$\lvert\eta\rvert < 0.7, p_t > 0.2$ GeV		1.85	6.5 μb	0.28
W^+W^-	7	Full 4π		29.6	39 fb	0.76
e^+e^-	7	ATLAS [93]	0.428 $\pm$ 0.035 $\pm$ 0.018	0.462	0.494 pb	0.94
$\mu^+\mu^-$	7	ATLAS [93]	0.628 $\pm$ 0.032 $\pm$ 0.021	0.738	0.789 pb	0.94
$\pi^+\pi^-$	13	ATLAS (Thesis) [94]	18.75 $\pm$ 0.048 $\pm$ 0.770	20	106 μb	0.19
$\mu^+\mu^-$	13	ATLAS [95]	3.12 $\pm$ 0.07 $\pm$ 0.14	3.35	3.49 pb	0.96
e^+e^-	1.96	CDF [96]	1.6 $\pm$ 0.5 $\pm$ 0.3	1.65	1.74 pb	0.95
e^+e^-	1.96	CDF [97]	2.88 $\pm$ 0.57 $\pm$ 0.63	3.24	3.37 pb	0.96
$\pi^+\pi^-$	1.96	CDF [98]	† $\pm$ † $\pm$ †	1.93	8.96 μb	0.22
$\mu^+\mu^-$	7	CMS [99]	3.38 $\pm$ 0.58 $\pm$ 0.21	3.88	4.09 pb	0.95
$\pi^+\pi^-_{EL}$	7	CMS [100]	26.5 $\pm$ 0.3 $\pm$ 5.12	11.5	57 μb	0.20
$\pi^+\pi^-_{SD}$	7	$\lvert y\rvert < 2, p_t > 0.2, M_1 < 5$ GeV		3.77	28.6 μb	0.13
$\pi^+\pi^-_{DD}$	7	$\lvert y\rvert < 2, p_t > 0.2, M_{1,2} < 5$ GeV		0.851	9.4 μb	0.09
$\pi^+\pi^-_{EL}$	13	CMS [101]	19.0 $\pm$ 0.6 $\pm$ 3.2	14.6	80.2 μb	0.18
$\pi^+\pi^-_{SD}$	13	$\lvert\eta\rvert < 2.4, p_t > 0.2, M_1 < 5$ GeV		4.22	34.2 μb	0.12
$\pi^+\pi^-_{DD}$	13	$\lvert\eta\rvert < 2.4, p_t > 0.2, M_{1,2} < 5$ GeV		0.903	11 μb	0.08

Table 5.1: LHC and Tevatron integrated fiducial cross section measurements compared with Graniitti using the screening loop σ_S and without σ_0. Processes are with elastic forward protons, unless otherwise stated. See the papers for the corresponding fiducial cuts. Note that data contains also non-exclusive background, which experiments have estimated and subtracted to obtain fully exclusive cross section in the case of lepton pairs. † The paper [98] includes a differential cross section in invariant mass, but no integrated cross section.

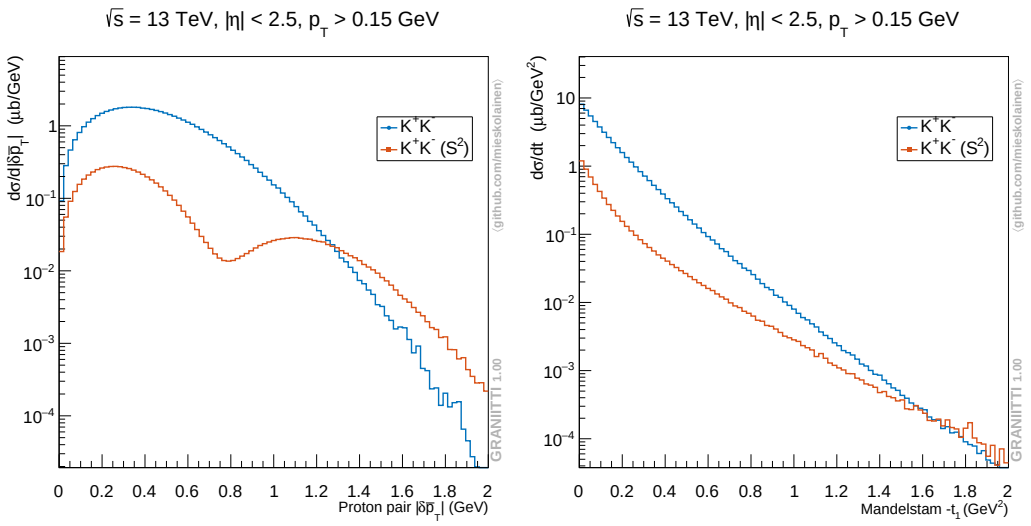

Figure 5.4: The screening loop effect on the K^+K^- continuum forward proton 'glueball filter' observable on the left and the Mandelstam $t_{1(2)}$ distribution on the right.

for the $2 \to N$ process with the corresponding measure $d\Pi_N$. For details, see Section 5.3. The largest absorption is obtained in Durham QCD model based production of a gluon pair gg. We see that lepton pair measurements have a systematically slightly lower cross section than the simulations. The simulations were done with k_t-EPA amplitudes, but this discrepancy is not explained by using the full tree-level $2 \to 4$ QED amplitude either, which is available in Graniitti. First one must remember that all photon-photon measurements in the table are non-exclusive and the exclusive cross section has been obtained by the collaborations via subtraction procedures. Regarding the semi-exclusive pion pair production, we see that our simulations are reasonably close to the experimental values obtained by CMS.

Forward dissociation

Central production together with single or double forward dissociation is implemented with exact skeleton kinematics, for details see Section 5.3. For dynamics the basic idea is that we can use the well known triple-pomeron limit $m_p^2/M^2 \ll 1$ and $M^2/s \ll 1$ of the forward system dissociation, which can be derived under the Mueller's generalized 3-body optical theorem [82]. This neglects low mass baryonic resonance structure and all internal structure fluctuations of the proton. In princi-

ple, one could think in terms of stochastic processes of partons. A coherent proton is controlled by a diffusive Gaussian process where only small changes at a given time interval are present. This results in a limited exponential type momentum transfer distribution. The dissociation represents an incoherent jump process, perhaps of Cauchy type, where a radical change may take over any finite time interval. Unifying these together under a relativistic QCD framework is a challenge.

Single and double dissociation

The triple leg unitarity diagrams give at the cross section level the well known results [82]

$$|\mathcal{A}_{SD}|^2 \sim [F(t) g_{ppP}]^2 \, g_{ppP} \, g_{PPP}(t) \left(\frac{s}{M_1^2}\right)^{\alpha_P(t_i)+\alpha_P(t_j)} \left(\frac{M_1^2}{s_0}\right)^{\alpha_P(0)} \tag{5.16}$$

$$|\mathcal{A}_{DD}|^2 \sim [g_{ppP} \, g_{PPP}(t)]^2 \left(\frac{s_0 s}{M_1^2 M_2^2}\right)^{\alpha_P(t_i)+\alpha_P(t_j)} \left(\frac{M_1^2}{s_0}\right)^{\alpha_P(0)} \left(\frac{M_2^2}{s_0}\right)^{\alpha_P(0)}, \tag{5.17}$$

where we replace $\alpha_P(t_i) + \alpha_P(t_j) = 2\alpha_P(t)$ and take the normalization scale $s_0 = 1$ GeV^2. In general, one could easily include here a sum over all $2^3 = 8$ different pomeron and reggeon combinations in the triple vertex with corresponding g_{ijk} couplings and trajectories. The i, j and k indicate the Pomeron positions in the diagrams, where k is the one with the inelastic cut. For simplicity, we however use only one triple pomeron coupling term with no t-dependence. The essential feature is that with double dissociation, there is no proton form factor dependent term and the sensitivity to the crucial triple pomeron coupling is squared. In the limit $t \to 0$, we obtain the most essential triple pomeron scaling law

$$\frac{d\sigma}{dM_1^2} \sim \frac{1}{(M_1^2)^{\alpha(0)}}. \tag{5.18}$$

Different reggeon-pomeron triple terms give different scaling laws driven by different trajectory combinations, respectively. The power law mass scaling gives clues in the direction of critical phenomena. The proton structure function parametrizations for the forward legs usually incorporate these Regge type scaling laws. The triple pomeron coupling has a fundamental role in studies of the Reggeon field theory in more fundamental terms, weak versus strong coupling and higher orders, parton branching picture and QCD. With eikonal screening we obtain approximately a value of $g_{PPP}/g_{ppP} \simeq 0.15$ with $g_{ppP} \simeq 8$ GeV^{-2} when no other triple leg terms are included. The value is small enough perhaps to consider that higher orders from

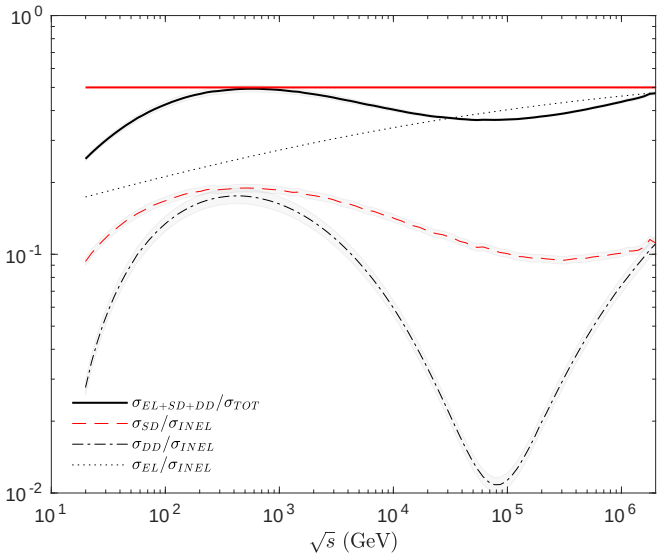

Figure 5.5: Inclusive cross section ratios and the Pumplin bound (solid red).

the full interacting Reggeon theory, which is an incomplete theory currently, can be neglected. It is well known that Eqs. 5.16 and 5.17 are not unitary, but scale with $s^{2\alpha_P(0)}$ and this is not compensated by the phase space which is for $2 \to 2$ given by $d\sigma/dt \sim 1/(16\pi s^2)$, combined together giving $\sim s^{2(\alpha_P(0)-1)}$. This is faster than the total cross section which grows like $s^{\alpha_P(0)-1}$ with a single pomeron exchange at large s.

To implement the eikonal screening for $2 \to 2$ single (SD) and double dissociation (DD) which restores unitarity, we use the same loop machinery as with central production, but here we need to recover the approximate equivalent bare amplitude by taking the square root and insert the pomeron signature factor phase – a crude first order inversion procedure which needs to be verified a posteriori. The integrated cross sections as a function of CMS energy are shown in Figure 5.6 with phase space integration cuts as indicated, to be able to compare with experimental data. In general, it is unknown both experimentally and theoretically what are the maximum forward mass or 'coherence' limits other than given by 4-momentum conservation, also the fragmentation gives irreducible smearing of the boundaries which we do not consider here.

An interesting feature is the energy evolution of the double dissociation cross

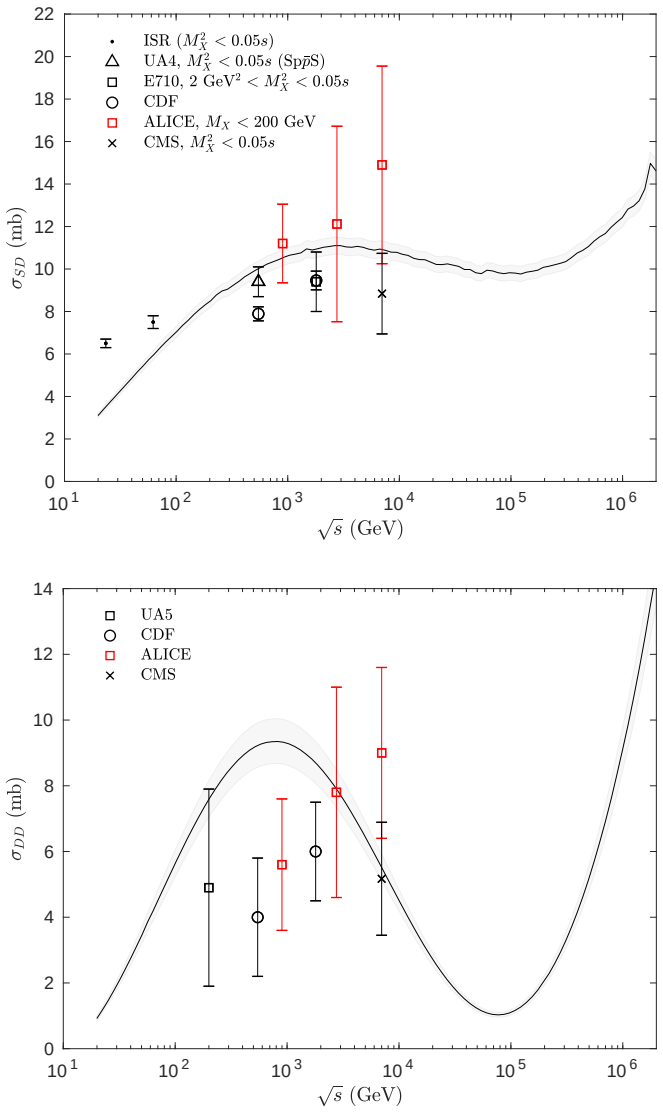

Figure 5.6: Eikonal screened single (forward + backward) and double diffraction integrated cross sections within phase space $|t| < 6$ GeV^2, $M_X^2 < 0.05s$ and $\Delta Y_{DD} > 3$, respectively. Data from [102–107].

section, which develops a strong 'bouncy' oscillation in our screening computation. Because of large uncertainties, higher precision data and collider energies up to $\sqrt{s} = 100$ TeV are needed to verify the behavior experimentally. The cross section ratios are shown in Figure 5.5, where the Pumplin unitarity bound [108] $\sigma_{EL+SD+DD}/\sigma_{TOT} < 1/2$ is not violated. The $\sigma_{EL}/\sigma_{INEL}$ ratio is seen to have near power law scaling towards value $1/2$ on the energy interval shown. What is known as the black disc (fully absorptive) limit corresponds to $\sigma_{EL} = \sigma_{INEL}$. Naturally, near the Planck scale or beyond we may expect some unexpected behavior.

To point out for clarity, we are not using the triple pomeron amplitudes in the central production dissociation, but pick up only the leading invariant mass dependence. We shall describe this next.

Central production with dissociation

The dissociative legs in the pomeron exchange central production are implemented via replacing the elastic vertex at amplitude level with an inelastic ansatz

$$g_{ppP}F(t) \mapsto [g_{ppP}g_{PPP}]^{\frac{1}{2}}\mathcal{F}(t, M^2), \tag{5.19}$$

where we take a coupling ansatz $[g_{ppP}g_{PPP}]^{\frac{1}{2}}$ motivated by the triple pomeron expressions. For the inelastic function $\mathcal{F}$ we use a minimal parametrization similar to the Donnachie-Landshoff proton structure function parametrization [109]

$$\mathcal{F}(t, M^2) = \left[\frac{s_0}{M^2}\frac{|t|}{|t|+a}\right]^{\frac{1}{2}\alpha_P(0)}, \tag{5.20}$$

where $a = 0.56$ GeV^2 and $s_0 = 1$ GeV^2, obeying the triple pomeron scaling. Once more measurements are available, this vertex and its normalization may be re-defined. In photon exchange processes, the inelastic vertex is described in Section 5.2.

We illustrate the fully elastic and dissociation cross sections obtained for the pion pair continuum production in Figure 5.7. The results depend on the process, so we may take the continuum amplitude as the most simple example. Before screening, we obtain the average ratios over a large range of energies

$$\frac{\sigma(\pi^+\pi^-_{SD})}{\sigma(\pi^+\pi^-_{EL})} \simeq 0.44, \quad \frac{\sigma(\pi^+\pi^-_{DD})}{\sigma(\pi^+\pi^-_{SD})} \simeq 0.39. \tag{5.21}$$

Interesting is to compare with the ratio $1/2$ of HERA results on dissociative photoproduction of vector mesons V with $\sigma_{\gamma p\to VX}(W_{\gamma p}) \simeq \frac{1}{2}\sigma_{\gamma p\to Vp}(W_{\gamma p})$. The HERA

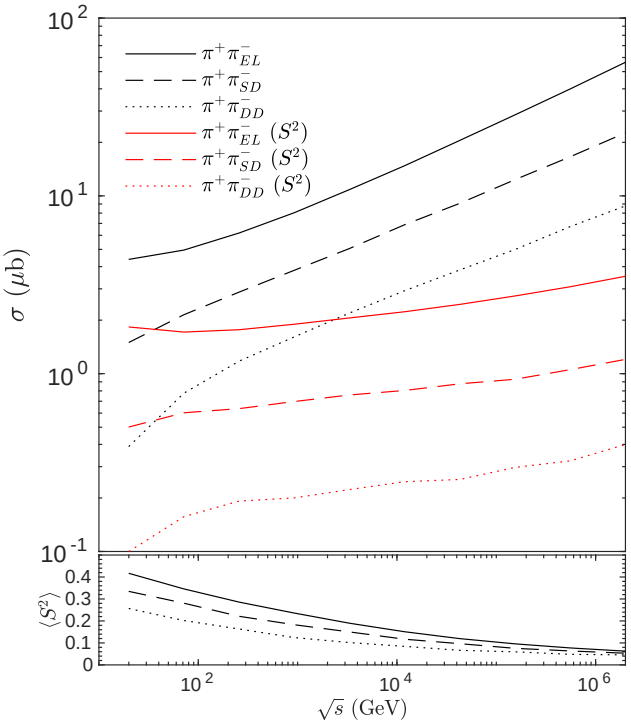

Figure 5.7: Continuum $\pi^+\pi^-$ pair production with elastic forward protons (EL), single dissociative (SD) and double dissociative (DD), without and with (S^2) the screening loop. In the lower figure the integrated screening ratio $\langle S^2 \rangle$ is shown. Fiducial cuts: $|\pi^\pm| < 0.9$ and $p_t(\pi^\pm) > 0.15$ GeV with forward system $M^2 < 0.05s$ and $\Delta Y_{DD} > 3$.

case is free from eikonal screening, thus it represents an interesting reference. After turning on the screening loop, we obtain the ratios

$$\frac{\sigma_S(\pi^+\pi^-_{SD})}{\sigma_S(\pi^+\pi^-_{EL})} \simeq 0.34, \quad \frac{\sigma_S(\pi^+\pi^-_{DD})}{\sigma_S(\pi^+\pi^-_{SD})} \simeq 0.29. \tag{5.22}$$

In Table 5.1 we have simulation results of semi-exclusive production for the LHC experiments. The simulation forward mass $M_{1,2}$ cuts given in the table of the dissociative production are approximations to the veto capabilities of the experiments. For a given experiment, there is a smooth forward mass (veto) efficiency function with asymptotics: $\epsilon(M) \to 0$ when $M \to 0$ and $\epsilon(M) \to \infty$ when $M \to \infty$. However, due to lack of instrumentation, the forward mass is not a direct experimental

observable at the LHC. Thus, this efficiency curve is obtained only through simulations and its verification is highly non-trivial, also because of significant detector material re-scattering effects. More precise effective cut estimates would require forward fragmentation by Pythia together with reliable effective fiducial pseudorapidity and transverse momentum cuts of the forward domain detectors obtained via Geant simulation. Anyway, we observe that there should be a significant dissociative contribution in the data after the forward vetoes. This contribution can be experimentally disentangled on a statistical basis by inspecting the transverse momentum spectrum.

Pomeron-Pomeron interactions

The double pomeron fusion production of two, four and six central final state continuum is provided, generalizing the $pp \to pM\bar{M}p$ meson pair amplitude described in [89, 110]. The simplest case is the two body continuum with the sub-t and sub-u channel amplitudes, here for a pion pair production

$$\begin{aligned}\mathcal{A}^{\hat{t}}_{pp\to p\pi^+\pi^- p} =& F(t_1) g_{ppP} \Delta_P(s_{13}, t_1) g_{\pi\pi P} F_M(\hat{t}) \Delta_M(\hat{t}) \times \\ & F_M(\hat{t}) g_{\pi\pi P} \Delta_P(s_{24}, t_2) g_{ppP} F(t_2),\end{aligned} \tag{5.23}$$

where the pomeron-pion coupling $g_{\pi\pi P}$ is obtainable from the total cross section fits. The pomeron propagator part with the signature factor is

$$\Delta_P(s,t) = \eta(\alpha_P(t), +) \left(\frac{s}{s_0}\right)^{\alpha_P(t)}, \tag{5.24}$$

with the normalization scale typically set $s_0 = 1$ GeV^2. The non-reggeized off-shell meson propagator is

$$\Delta_M(\hat{t}) = \frac{1}{\hat{t} - M_0^2}, \tag{5.25}$$

where M_0 is the pion mass. Here we use a typical linear pomeron trajectory parametrization and evaluate the signature factor with $\eta(t) \simeq \eta(0) e^{-i\frac{\pi}{2}\alpha' t}$ limit to avoid signature poles encountered with the linear trajectory at high $|t|$. The sub-u channel amplitude is obtained by crossing $3 \leftrightarrow 4$ and thus $\hat{t} \leftrightarrow \hat{u}$.

The kinematic invariants are $\hat{t} = (q_1 - p_3)^2, \hat{u} = (q_1 - p_4)^2$ and $s_{ij} = (p_i + p_j)^2$, where $i = 1, 2$ denote the forward systems. The 4-momentum transfer squared are $t_1 = (p_A - p_1)^2, t_2 = (p_B - p_2)^2$ and the total CMS energy squared is $s = (p_A + p_B) = (p_1 + p_2 + p_3 + p_4)^2$. For the four and six body central states we need a set of

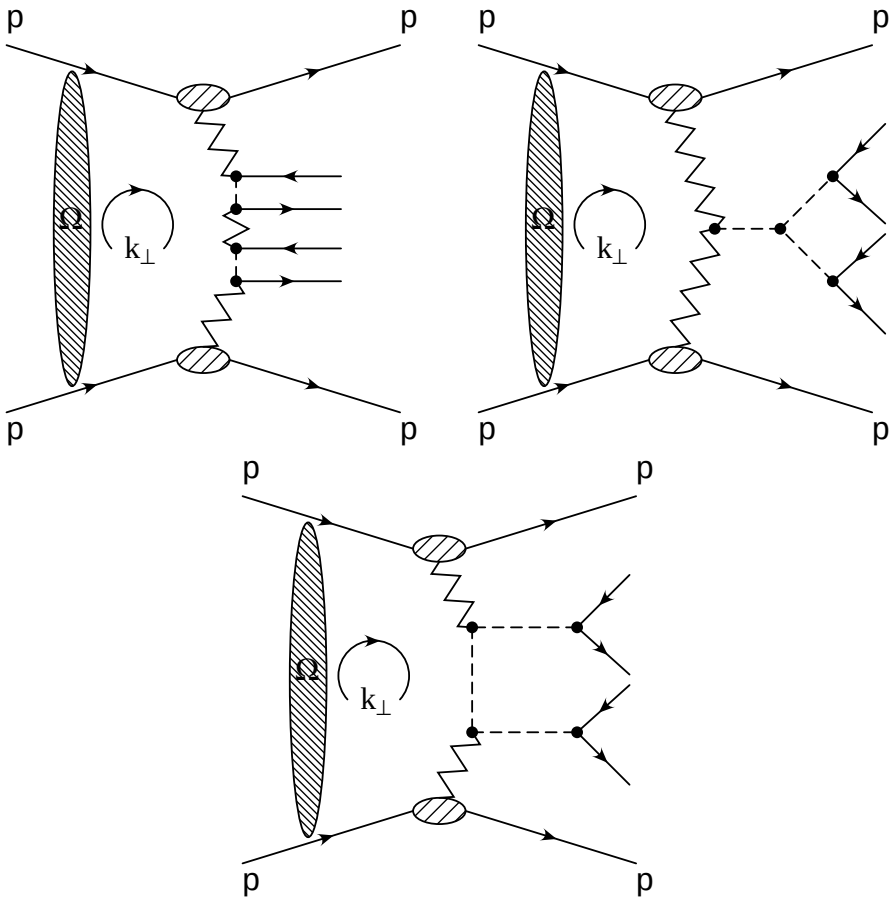

Figure 5.8: A pomeron-pomeron 4-body continuum amplitude on the left, resonance production with intermediate states on the right and 2-body continuum with intermediate states at the bottom, all with eikonal screening.

generalized invariants. For the straightforward details, we refer to the code which calculates these and the amplitudes algorithmically. Enumerating all permutations of charged particle final state legs gives 2, 16 and 288 sub-amplitudes growing like Cayley-Menger determinant absolute coefficients. Some of the amplitude topologies are illustrated in Figure 5.8.

For the off-shell meson sub-form factor $F_M(\hat{t})$, not to be mixed with the proton form factor, there are several options already familiar from [89], such as an exponen-

tial, Orear-like and a power law

$$F_M^{\mathrm{exp}}(\hat{t};\beta) = \exp\left(-\beta|t'|\right) \tag{5.26}$$

$$F_M^{\mathrm{Orear}}(\hat{t};\kappa_1,\kappa_2) = \exp\left(-\kappa_2\sqrt{|t'|+\kappa_1^2}+\kappa_1\kappa_2\right) \tag{5.27}$$

$$F_M^{\mathrm{pow}}(\hat{t};\kappa) = \frac{1}{1-t'/\kappa}, \quad \text{with} \quad t' \equiv \hat{t} - M_0^2. \tag{5.28}$$

By default, we use the power law which seems to work quite well. These form factors *do* contribute significant differences, effectively encapsulating unknown non-perturbative effects in QCD, which in the perturbative limit typically has power law behavior. We emphasize that the continuum amplitudes here have no explicit active helicity degrees of freedom, but the produced angular distributions are still far from isotropic $J = 0$. This is due to the amplitude structure. For a scenario where also the continuum amplitudes incorporate helicities, see the Tensor pomeron model amplitudes in Section 5.2. For the spin analysis purposes, we provide a generation mode of the crucial pure $J = 0$ continuum with exponential or limited $t_{1,2}$-distributions for any number of final states produced by the isotropic decay.

The production of resonances interfering with the continuum is described with simple relativistic Breit-Wigner poles

$$\Delta_{BW}^R(m^2) = \frac{1}{m_R^2 - m^2 - im_R\Gamma}, \tag{5.29}$$

with the fixed or running width Γ and the pole mass m_R. The amplitude in the pomeron-pomeron fusion for the production of a resonance R decaying to particles 3 and 4 is

$$\begin{aligned}\mathcal{A}^{\hat{s}}_{pp\to pR(\to 34)p} = {} & F(t_1)\Delta_P(s_1,t_1)g_{ppP}\Delta_{BW}^R(m^2)\times \\ & \mathcal{V}_{PP}^{R\to 34}(q_1,q_2,p_3,p_4)\Delta_P(s_2,t_2)g_{ppP}F(t_2),\end{aligned}$$

The kinematic variables are standard Lorentz scalars, discussed in Appendix B.2. The pomeron-resonance-pomeron vertex has an unknown effective functional structure, which we write as

$$\begin{aligned}&\mathcal{V}_{PP}^{R\to 34}(q_1,q_2,p_3,p_4) \\ &= \left[\frac{s_0}{(q_1+q_2)^2}\right]^{\omega}\mathcal{A}_{PP\to R}(q_1,q_2)\mathcal{A}_{R\to 34}(p_3,p_4)F_R(q_1,q_2).\end{aligned} \tag{5.30}$$

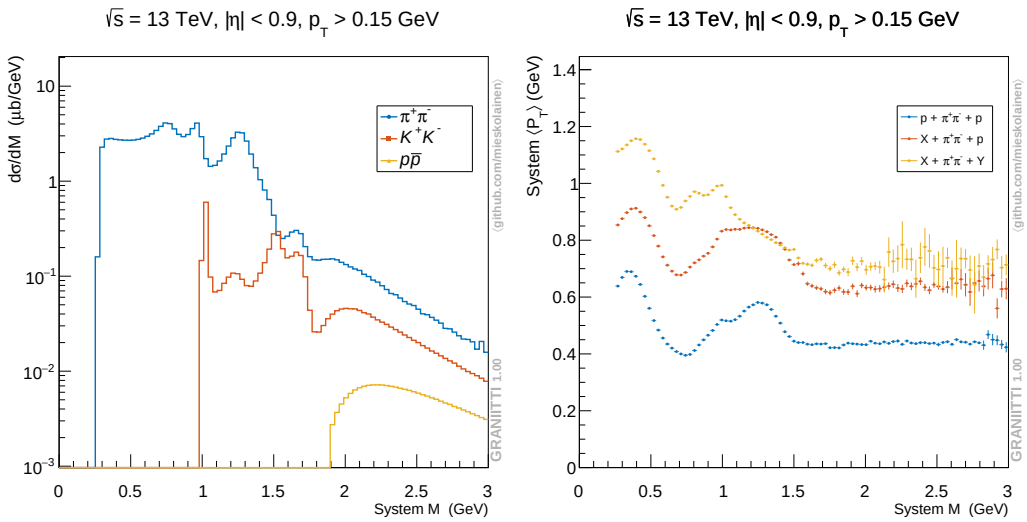

Figure 5.9: The invariant mass spectrum of $\pi^+\pi^-$, K^+K^- and $p\bar{p}$ pair on the left and the mean transverse momentum of the $\pi^+\pi^-$ spectrum with elastic, single and double dissociative production on the right. A constant $\langle S^2 \rangle \equiv 0.15$ applied.

The terms $\mathcal{A}_{PP\to R}$ and $\mathcal{A}_{R\to 34}$ encapsulate the production and the decay couplings and helicity dependence, respectively. The resonance production couplings $g_{PP\to R}$ embedded in $\mathcal{A}_{PP\to R}$ are free parameters (one per resonance) which we allow to be complex for maximal flexibility, to set up the relative complex phase between the continuum and resonance amplitudes. The rest of the production parameters for non-scalar resonances belong in our description to the resonance spin polarization density matrix. The vertex dependence in terms of the invariant mass squared $(q_1 + q_2)^2$ is parametric and the value $\omega = 1/2$ seems suitable. Technically, this is related to the resonance form factor

$$F_R(q_1, q_2) = \exp\left(-\left[(q_1 + q_2)^2 - m_R^2\right]^2 / \Lambda_0^4\right), \quad \Lambda_0 \approx 1.0 \text{ GeV} \tag{5.31}$$

which is responsible for the finite size of the resonance. These both together modify the spectral shape beyond the effect of rising low-mass phase space, the Breit-Wigner propagator, Pomeron propagators and interference with the continuum amplitude. Again, alternative form factors are clearly possible.

Now we see very clearly that it is non-trivial to say which are the 'true' mass and width parameters of the composite resonances here because they depend effectively

on the production process, a well known feature. Figure 5.9 shows the invariant mass spectrum and average transverse momentum for $\pi^+\pi^-$, K^+K^- and $p\bar{p}$ pairs. We emphasize that the system transverse momentum allows one to gain control of the dissociative contribution, assuming that the elastic central production t-distribution is properly understood and controlled by forward proton measurements. The elastic case is driven by the proton form factors which are here fixed by the eikonal pomeron fits and the rest of the t-dependence comes from the pomeron slope α' dependent parts of Eq. 5.24.

Minimal spin pomeron

The polarized decay part $\mathcal{A}_{R\to 34}$ for non-scalar resonances is driven by the spin polarization density matrix of the resonance and Jacob-Wick helicity $1 \to 2$ amplitudes, details of this formalism are given in Section 5.2. For the production part $\mathcal{A}_{PP\to R}$, we first assume a *minimal effective spin* for the fusing pomerons compatible with the basic conservation laws. We take the pomeron spin to be $J = 0$ for the production of scalar or tensor meson resonances, also in photoproduction, and take $J = 1$ for the production of pseudoscalars. We have also other implementations in the code such as the sliding helicity trajectories, as we discuss in Appendix B.1. Our minimal choice introduces in the most resonance spin-parity cases no additional free parameters.

After fixing the spin of pomerons, we use the helicity amplitudes in $2 \to 1$ direction for the two pomeron fusion. This generates the forward proton azimuthal distribution $\Delta\varphi$ modulation which is dependent on the spin polarization matrix elements. The feasibility of this computational trick depends on the chosen Lorentz rest frame because these explicitly *non-covariant* amplitudes are parametrized by a frame-dependent spin polarization density matrix. Some rotated frames where pomerons have fixed $(\cos\theta, \varphi)$ event-by-event, such as the Gottfried-Jackson with pomerons back-to-back on the z-axis, are not directly suitable, neither is the helicity frame resulting (seemingly) in a too strong system rapidity dependence. We found out that a suitable generation frame is the Collins-Soper frame, which gives results in a good agreement with the preliminary ATLAS+ALFA data with $\Delta\varphi < \pi/2$ and $\Delta\varphi > \pi/2$ cuts [94]. This simulation is shown in Figure 5.10, also directly relevant to the upcoming CMS+TOTEM measurements. We see also that the t-acceptance cut results in a strong suppression of the photoproduced ρ^0, as expected. When the pomeron spin is not zero, one should couple the proton legs with pomerons due to the pomeron polarization information. Our machinery is capable of this, but we encounter it only in the case of pseudoscalar resonances because of our minimal description. The pseudoscalar case is calculated in a faster way by using the results

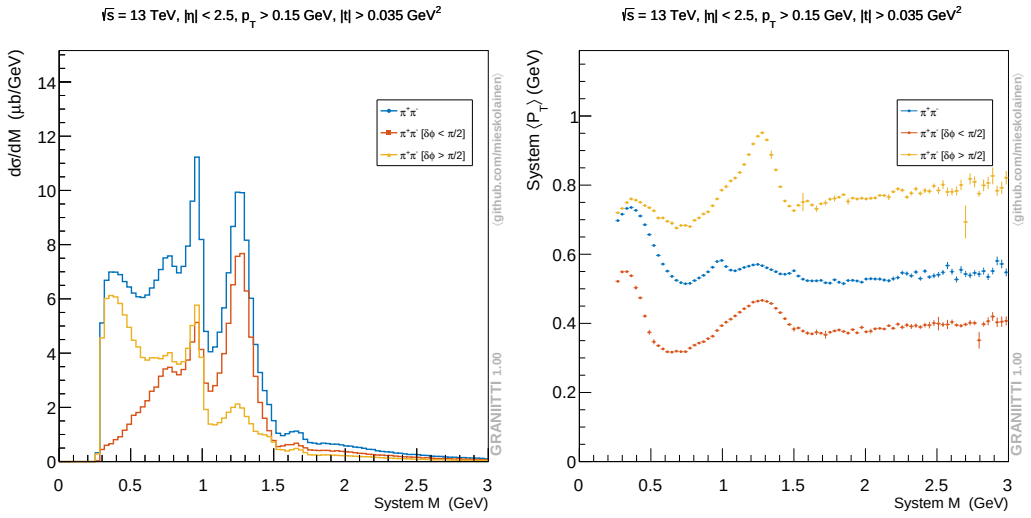

Figure 5.10: The exclusive $\pi^+\pi^-$ pair production without and with forward proton transverse plane angle cuts on the left. The mean transverse momentum with the same transverse plane cuts on the right. A constant $\langle S^2 \rangle \equiv 0.15$ applied.

discussed in [111]. People interested in *covariant* $2 \to N$ spin-dependent amplitudes may generate events using the tensor pomeron model described in Section 5.2.

The parameters of the spin polarization density can be changed to arbitrary values, but currently we take a diagonal ansatz of $|J_z| = \pm 1$ in photoproduction and $|J_z| = \pm 2$ for the tensor resonances such as $f_2(1270)$. This discussion is intimately related to the glueball filter introduced in [64]. Finally, the total amplitude is a coherent sum over the continuum and resonances

$$\mathcal{A}^{\mathrm{tot}} \equiv \mathcal{A}^{\hat{t}} + \mathcal{A}^{\hat{u}} + \sum_R \mathcal{A}_R^{\hat{s}}. \tag{5.32}$$

The highly modular structure of our code and the input engine allows 'experimenting' with many of the phenomenological aspects of the scattering amplitudes, or plugging in completely new ones. Interesting would be to see if models such as 'dual amplitudes' of early strings would provide useful input here for the resonance coupling systematics [112]. A full theory would take into account also the analyticity and unitarity of the S-matrix and derive all the resonance couplings, something which is currently much beyond even the most radical amplitude technology.

Photoproduction

We have a (k_t-EPA)-pomeron based photoproduction of vector mesons, with a purpose of working as a suitable semi-reference process between the measurements done with and without forward proton tagging. In future we may implement dipole picture based models, such as variants of Golec-Biernat and Wüsthoff [113]. Due to the typical experimental t-acceptance at the LHC, the photoproduced vector mesons should disappear from the spectrum, but be well visible without the proton tagging. This seems to be the case in data as seen in [101] and [94]. The amplitude for the photoproduction of a vector meson V with a spin polarization dependent decay is

$$\mathcal{A}_{pp\to pV(\to 34)p} = \mathcal{A}^{V\to 34}_{\gamma P\to V} + \mathcal{A}^{V\to 34}_{P\gamma\to V}, \tag{5.33}$$

where

$$\mathcal{A}^{V\to 34}_{\gamma P\to V} = \left[\frac{1}{\xi_1} f^{EL}_{\gamma}(\xi_1, \mathbf{q}_{t,1})\right]^{\frac{1}{2}} \Delta^{V}_{BW}(m^2)\mathcal{V}^{V\to 34}_{yP}(q_1, q_2, p_3, p_4)\mathcal{A}^{V}_{P}(s_2, t_2). \tag{5.34}$$

The amplitude $\mathcal{A}_{P\gamma}$ is obtained by permuting the variables with $1 \leftrightarrow 2$ and the central vertex $\mathcal{V}$ is

$$\mathcal{V}^{V\to 34}_{yP}(q_1, q_2, p_3, p_4) = \mathcal{A}_{\gamma P\to V}(q_1, q_2)\mathcal{A}_{V\to 34}(p_3, p_4)F_R(q_1, q_2). \tag{5.35}$$

For more details about the photon flux f_{EL} and kinematic variables see Section 5.2. The relative sign of Eq. 5.33 would be negative in the case of proton-antiproton collisions, due to the anti-symmetric initial state. The pomeron side factor combining the s_2 and t_2-dependence is

$$\mathcal{A}^{V}_{P}(s_2, t_2) = \Delta_P(s_2, t_2)\exp(b_0 t_2/2). \tag{5.36}$$

See also that $s_2 \equiv W^2_{\gamma p}$ is the typical notation in the deep inelastic scattering context. The normalization scale $s_0 = W^2_0 = 90^2$ GeV^2 in the pomeron propagator of Eq. 5.24 is the most typical scale to fix the parameters. Note that the energy dependence of the subprocess is experimentally at larger energies approximately [114]

$$\sigma(\gamma p \to \rho^0 p) \propto W^{0.22}_{\gamma p} \tag{5.37}$$

$$\sigma(\gamma p \to J/\psi\, p) \propto W^{0.65}_{\gamma p}, \tag{5.38}$$

which motivates in the literature the large intercept ~ 0.3 hard pomeron (BFKL ladder) type production interpretation of J/ψ and Υ, where as for low mass vectors the soft pomeron seems suitable.

The gamma-pomeron-vector meson coupling and the pomeron side exponential b_0-slope have been fixed with HERA data. The slope parameters b_0 are typically ~ 11 GeV^{-2} for ρ^0 production down to $\sim 4-5$ GeV^{-2} for J/ψ, reflecting the intrinsic transverse size of the vector meson $q\bar{q}$ dipole. These experimental fit values combine the effect of both the vector meson and the proton form factor. The photon side has a much steeper t-dependence than the pomeron side, which is included in our description. The main expected difference between photoproduction and speculative odderon-pomeron fusion is thus in the transverse momentum dependence, but possibly also in the polarization of the produced vector mesons, which is easily changed in the simulation. Also, we have simple odderon-pomeron amplitudes within a simplified description. We take the odderon simply as odd signature pomeron with unknown couplings to be fixed.

Decay couplings

The effective decay coupling for two-body decays $M_R \to m_1 + m_2$ is calculated according to the branching ratios $\mathrm{BR}_i \equiv \Gamma_i/\Gamma$ imported from the PDG and the standard partial decay width formula which factorizes, in the $1 \to 2$ case, between the phase space and the decay matrix element squared. The partial decay width is the well known

$$\Gamma_i = \frac{1}{S}\frac{\sqrt{\lambda(M_R^2, m_1^2, m_2^2)}}{16\pi M_R^3}|\mathcal{M}_D|^2, \tag{5.39}$$

where the $\sqrt{\lambda}$ is the standard Källen triangle function and S is the statistical symmetry factor. That is, we simply invert the relation to find out the effective decay coupling $|\mathcal{M}_D|^2 \sim |g_D|^2$ given the measured branching ratios. The full decay matrix element is non-perturbative and unknown here for the f-mesons and glueballs but may be estimated under certain frameworks such as holography. For non-scalar decays, one needs to take into account the spin related normalization factors in the machinery in Section 5.2. To obtain higher precision, we could calculate this using our generic phase space sampling functions which integrate also phase space volumes, more suitable for large width resonances and near threshold behavior. The machinery allows one to add arbitrary many new resonances and their decays. We provide arbitrary deep decay chains according to the phase space but also with (cascaded) spin correlations initiated by the resonance amplitude. The cascaded spin correlations require the intermediate decay ls-couplings as an input. A flexible decay chain machinery is highly important for many experimental analyses, which need to test different hypotheses and evaluate the significance of various 'feed-down' contribu-

tions, for example.

Photon-Photon interactions

Two photon interactions are generated according the k_t-EPA Equivalent Photon Approximation formalism for the photon fluxes from protons, with both elastic and inelastic fluxes at the cross section level [115, 116]

$$f_\gamma^{EL}(\xi, \mathbf{q}_t) = \frac{\alpha}{\pi}\left[(1-\xi)\Delta^2 \frac{4m_p^2 G_E^2(Q^2) + Q^2 G_M^2(Q^2)}{4m_p^2 + Q^2} + \frac{\xi^2}{4}\Delta G_M^2(Q^2)\right]$$
$$f_\gamma^{IN}(\xi, \mathbf{q}_t, M_X^2) = \frac{\alpha}{\pi}\left[(1-\xi)\Delta^2 \frac{F_2(x_{Bj}, Q^2)}{Q^2 + M_X^2 - m_p^2} + \frac{\xi^2}{4x_{Bj}^2}\Delta \frac{2x_{Bj}F_1(x_{Bj}, Q^2)}{Q^2 + M_X^2 - m_p^2}\right]$$
$$\text{with} \quad \Delta \equiv \frac{\mathbf{q}_t^2}{\mathbf{q}_t^2 + \xi(M_X^2 - m_p^2) + \xi^2 m_p^2}, \tag{5.40}$$

where $\xi = 1 - p_z'/p_z$ is the longitudinal momentum loss of the proton, $\mathbf{q}_t$ the photon transverse momentum, $Q^2 \equiv -t$ the 4-momentum transfer squared, $x_{Bj} = Q^2/(Q^2 + M_X^2 - m_p^2)$ is the Bjorken-x identically one for the elastic case and M_X^2 is the forward system invariant mass. These fluxes are then matched with the exact $2 \to N$ phase space construction taking into account the kinematic factors by a transform at the cross section level

$$f_\gamma \mapsto 16\pi^2 [\xi \mathbf{q}_t^2]^{-1} f_\gamma, \tag{5.41}$$

which we have verified against the full QED $pp \to pl^+l^-p$ tree level amplitude.

The coherent proton electromagnetic form factors in Sachs form [117] are G_E and G_M. By construction, the linear relation between Sachs and F_1 (Dirac) and F_2 (Pauli) form factors can be written as

$$\begin{pmatrix} G_E \\ G_M \end{pmatrix} = \begin{pmatrix} 1 & -\tau \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} \Leftrightarrow \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} \frac{1}{\tau+1} & \frac{\tau}{\tau+1} \\ -\frac{1}{\tau+1} & \frac{1}{\tau+1} \end{pmatrix} \begin{pmatrix} G_E \\ G_M \end{pmatrix}, \tag{5.42}$$

where $\tau = Q^2/(4m_p^2)$. A simple dipole parametrization is used

$$G_E(Q^2) = G_D(Q^2) = \frac{1}{(1 + Q^2/\lambda^2)^2} \tag{5.43}$$
$$G_M(Q^2) = \mu_p G_D(Q^2), \tag{5.44}$$

where the proton magnetic moment $\mu_p = 2.7928$ in units of the nuclear magneton $\mu_N = e\hbar/(2m_p)$ and $\lambda^2 = 0.71$ GeV^2. The normalization here gives $F_1(0) = 1$

and $F_2(0) = 1.7928$. In addition, we have included more complex but still simple parametrization from [118].

In the inelastic case, the proton structure functions are $F_2(x, Q^2)$ and $F_1(x, Q^2)$, for which we use some classic parametrizations, to be plug-in replaced easily by more relevant up-to-date external libraries or implemented by the user. In the parton model the Callan-Gross relation holds

$$2xF_1(x) = F_2(x), \tag{5.45}$$

due to spin-1/2 quarks, but in QCD with gluons a longitudinal structure function component is present

$$F_L(x, Q^2) \equiv (1 + \frac{4x^2 m_p^2}{Q^2})F_2(x, Q^2) - 2xF_1(x, Q^2), \tag{5.46}$$

which may be extracted from the scaling violations. The longitudinal component is especially relevant at small values of Bjorken-x where gluon density rises steeply. The terminology between transverse and longitudinal stems from the deep inelastic scattering and the corresponding virtual photon polarization component cross sections. With high enough Q^2, for the F_2 here one could use the QCD evolved parton density description

$$F_2(x, Q^2) = x \left(\sum_{i \in u_v, d_v} e_i^2 f_i(x) + \sum_{i \in u_s, d_s, s_s} e_i^2 [f_i(x) + \bar{f}_i(x)], \right), \tag{5.47}$$

with $e_i^2 = 4/9\,(1/9)$ for up (down) type quarks, where the parton densities are readily available through the LHAPDF 6 library interface [119]. The longitudinal structure function F_L can be related to gluon densities within pQCD, an interesting topic also algorithmically, for a recent work see [120].

To cross check the k_t-EPA implementation differentially, we have implemented the $pp \to p\ell\bar{\ell}p$ tree level $2 \to 4$ full QED process with the standard covariant current

$$J^\mu = ie\bar{\nu}(p', \lambda')[\gamma_\mu F_1(Q^2) + \frac{i\sigma_{\mu\nu}}{2m_p}(p - p')^\nu F_2(Q^2)]\nu(p, \lambda), \tag{5.48}$$

where the term with F_1 is the helicity λ conserving part and the term with F_2 is the helicity flip (non-conserving) part and $\sigma_{\mu\nu} = \frac{i}{2}(\gamma_\mu\gamma_\nu - \gamma_\nu\gamma_\mu)$. This implementation also provides proper distributions e.g. for very low invariant masses of the lepton pair system, where the k_t-EPA + on-shell $\gamma\gamma \to X$ matrix elements are on the edge of their validity.

Currently, we have included on-shell matrix elements for the lepton pair and W^+W^- production with helicity amplitudes imported from MadGraph5 C++ export [121]. Because our kinematics construction is exact $2 \to N$, the photon kinematics have always small but finite $q_i^2 < 0$, but MadGraph $\gamma\gamma \to X$ amplitudes assume $q_i^2 = 0$. We correct this by transforming the initial state photons to the closest point at light cone which is found by Lagrange multipliers and iterate the final state kinematics so that energy-momentum is conserved and amplitudes can be safely evaluated. This procedure might be optimized in the future.

In addition, we have also $j = 1/2$ monopolium pair production and monopolium bound state $J = 0$ resonance production with pure Dirac and velocity-dependent couplings, as simple scenarios of fundamental magnetic monopoles. As magnetic monopoles are strongly coupled, intrinsically non-perturbative and currently lacking rigorous field theory framework, the QED matrix element replacement represents a somewhat uncontrolled approximation, but still useful. The bound state modeling is based on a simple Schrödinger equation type solution.

Durham QCD model

For the Durham QCD (KMR) model [122], we include the numerical gluon loop with spin-parity projection and a generalized gluon pdf transformation which includes the Shuvaev transform and Sudakov radiation suppression. The main interest for us here is the transition region from the low mass Regge domain to this QCD domain. We go now through the formulation, for more details see [123, 124]. The formulation starts with the amplitude at parton level for $qq \to q + X + q$ with a gluon loop, which is most easily derived under the high energy eikonal forward quark vertex limit

$$-i\bar{u}(p',\lambda')ig_sT^a_{ij}\gamma^\mu u(p,\lambda) \to -2g_sp^\mu T^a_{ij}\delta_{\lambda,\lambda'}, \tag{5.49}$$

where $T^a_{ij} = \lambda^a_{ij}/2$ is the $SU(3)$ generator matrix element $\langle i|T^a|j\rangle$ and color indices run $a = 1,\ldots,8$ and $i,j = 1,2,3$. The amplitude will be dominantly imaginary in the forward limit and the imaginary part of the loop amplitude can be obtained most easily with Cutkosky cutting rules [125], which replace propagators by delta functions. The Durham model is illustrated in Figure 5.11, where the Cutkosky line goes vertically through the gluon loop.

We denote the quark momentum with color indices in the parenthesis

$$\text{Left side of the cut: } p_1(i) + p_2(j) \to p_1(m) + p_2(n) \tag{5.50}$$

$$\text{Right side of the cut: } p_1(m) + p_2(n) \to p_1'(i) + p_2'(j), \tag{5.51}$$

where the color is oriented along the quark lines so that the system X is color singlet and the amplitude will be compatible at the proton level. The sreening gluon carries momentum Q with color c on the left side of the cut and the fusing gluons $q_1(a), q_2(b)$ on the right side. Now writing these down gives [123]

$$\mathrm{Im}\,\mathcal{A} = \underbrace{\frac{1}{2}}_{\text{cut-rule}} \times \underbrace{2}_{\#\text{diagrams}} \times \int d\Pi_2 \frac{2g_s p_1^\rho \cdot 2g_s p_{2\rho}}{Q^2} \frac{2g_s p_1^\mu}{q_1^2} \frac{2g_s p_2^\nu}{q_2^2} \times \tag{5.52}$$

$$V_{\mu\nu}^{ab} T_{mi}^c T_{im}^a T_{nj}^c T_{jn}^b \, \delta\left((p_1 - Q)^2\right) \delta\left((p_2 + Q)^2\right). \tag{5.53}$$

At this point we take the integral over transverse momentum space in the eikonal limit

$$\int d\Pi_2 \equiv \frac{1}{2s} \int \frac{d^2\mathbf{Q}_t}{(2\pi)^2}, \tag{5.54}$$

which can be overall justified by Sudakov decomposition, and we will change the momentum variables to transverse variables. The average and sum over colors at amplitude level gives

$$\frac{1}{N_C^2} T_{mi}^c T_{im}^a T_{nj}^c T_{jn}^b = \frac{1}{N_C^2} \mathrm{Tr}[T^c T^a] \mathrm{Tr}[T^c T^b] = \frac{1}{4N_C^2} \delta^{ca} \delta^{cb} = \frac{\delta^{ab}}{4N_C^2} \to \frac{2}{9}, \tag{5.55}$$

where the last step is obtained, when the sub-amplitude gives equal $N_C^2 - 1$ contributions such as the SM Higgs production. The normalization color factor is $T_R = 1/2$ from $\mathrm{Tr}[T^a T^b] = T_{ij}^a T_{ij}^b = T_R \delta^{ab}$ associated with the gluon splitting into quark pair, $C_F = (N_C^2 - 1)/(2N_C) = 4/3$ is the color factor with gluon emission from quarks $\sum_a T_{ik}^a T_{kj}^a = \sum_a (T^a T^a)_{ij} \equiv C_F \delta_{ij}$ and the gluon splitting into gluon color factor $f_{acd} f_{bcd} = C_A \delta_{ab}$ with $C_A \equiv N_C = 3$. The generators are traceless $\mathrm{Tr}[T^a] = \sum_{i=1}^{N_C} T_{ii}^a = 0$.

Kinematics and coupling factors give us

$$\frac{1}{2} \times 2 \times \frac{1}{2s} \frac{1}{(2\pi)^2} \times \frac{s}{2} \times (2\sqrt{4\pi\alpha_s})^4 = 16\alpha_s^2. \tag{5.56}$$

Combined together with color factors we get

$$\frac{\delta^{ab}}{4N_C^2} \times 16\alpha_s^2 \xrightarrow{V_{\mu\nu}^{ab} \sim \delta^{ab}} \frac{N^2 - 1}{N^2} 4\alpha_s^2, \tag{5.57}$$

where the last step is in the case of SM Higgs like amplitude.

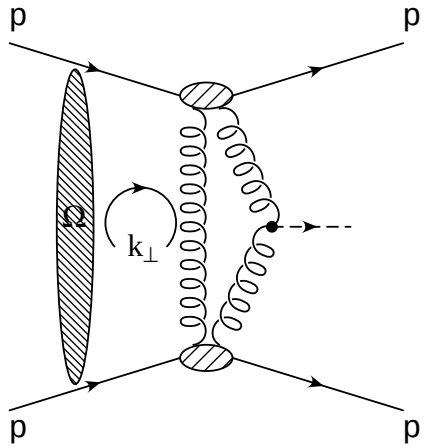

Figure 5.11: Durham QCD model with eikonal screening.

The vertex contraction is manipulated into fusing gluon form by setting $q_i = x_i p_i + q_{t,i}$ and $2q_1 \cdot q_2 \simeq M_X^2$

$$p_1^\mu p_2^\nu V_{\mu\nu}^{ab} \simeq \frac{q_{t,1}^\mu}{x_1} \frac{q_{t,2}^\nu}{x_2} V_{\mu\nu}^{ab} \xrightarrow{s x_1 x_2 \simeq M_X^2} \frac{s}{M_X^2} q_{t,1}^\mu q_{t,2}^\nu V_{\mu\nu}^{ab}, \tag{5.58}$$

where one could use gauge invariance $q_1^\mu V_{\mu\nu}^{ab} = q_2^\nu V_{\mu\nu}^{ab} = 0$. The contraction with the vertex is as with external polarization vectors $\epsilon \sim q$, which means a transverse polarization with $\epsilon_1 = -\epsilon_2$ from $Q_t = -q_{1,t} = q_{2,t}$ in the pure forward limit $p'_{t,1,2,} \to 0$ of outgoing quarks giving the $J_z = 0$ selection rule.

Now we denote the color averaged sub-amplitude as [126]

$$\langle \mathcal{M} \rangle \equiv \frac{2}{M_X^2} \frac{1}{N_C^2 - 1} \sum_{a,b=1}^{N_C^2-1} \delta^{ab} q_{1,t}^\mu q_{2,t}^\nu V_{\mu\nu}^{ab}. \tag{5.59}$$

The total result matching the derivation above can be written as

$$\frac{\mathrm{Im}\,\mathcal{A}}{s} \simeq C_F^2 \alpha_s^2 \int \frac{d^2\mathbf{Q}_t}{\mathbf{Q}_t^2 \mathbf{q}_1^2 \mathbf{q}_2^2} \langle \mathcal{M} \rangle. \tag{5.60}$$

What one notices, is the infrared divergence in the loop Q_t^2 which is to be tamed by the Sudakov resummation.

Now, the description above was at the parton level. At the hadron level, we need the generalized gluon 'Durham flux' from the proton [127]

$$f_g(x,x',Q_t^2,\mu^2) = \frac{\partial}{\partial \ln Q_t^2}\left[\sqrt{T(Q_t^2,\mu^2)}G(\frac{x}{2},\frac{x}{2},Q_t^2)\right], \tag{5.61}$$

where the generalized (skewed) gluon pdf is obtained by a 'Shuvaev transform' of the standard integrated gluon pdf $g(x,Q^2)$ with an integral method [128]

$$G(\frac{x}{2},\frac{x}{2},Q_t^2) = \frac{4x}{\pi}\int_{\frac{x}{4}}^1 dy\, g\left(\frac{x}{4y},Q_t^2\right)(1-y)^{1/2}y^{1/2}, \tag{5.62}$$

and the Sudakov suppression vetoing real radiation is

$$T(Q_t^2,\mu^2) = \exp\left(-\int_{Q_t^2}^{\mu^2}\frac{dk_t^2}{k_t^2}\frac{\alpha_s(k_t^2)}{2\pi}\int_0^{1-\Delta}dz\left[zP_{gg}(z)+\sum_{q\in\text{flavors}}P_{qg}(z)\right]\right), \tag{5.63}$$

where $P_{gg}(z)$ and $P_{qg}(z)$ are leading order DGLAP splitting kernels. In the integral the upper bound $\Delta = k_t/\mu$ is taken as described in detail in [124]. For the discussion of single and double logarithmic terms and the origin of the derivative in Eq. 5.61, see [127]. For completeness, the splitting kernels are for gluon $P_{g\leftarrow g}$ and quark $P_{q\leftarrow g}$ emissions

$$P_{gg}(z) = 2C_A\left[\frac{z}{(1-z)_+}+\frac{1-z}{z}+z(1-z)\right]+\delta(1-z)\frac{11C_A-4n_fT_R}{6} \tag{5.64}$$

$$P_{qg}(z) = T_R[z^2+(1-z)^2], \tag{5.65}$$

where the 'plus prescription' is

$$\int_0^1 dz\,\frac{f(z)}{(1-z)_+} = \int_0^1 dz\,\frac{f(z)-f(1)}{1-z}, \quad \text{with} \quad (1-z)_+ = 1-z, \quad \text{for } z<1, \tag{5.66}$$

so formally the soft gluon divergence at $z=1$ cancellation relies on $f(1)$. The integration and differentiation are done numerically, the results being cached into look-up tables and interpolated. The Sudakov suppression is illustrated in Figure 5.12.

Finally at the proton level, the scattering amplitude for the total process is given by

$$\mathcal{T} = \frac{\text{Im}\,\mathcal{A}}{s} = \pi^2\int\frac{d^2\mathbf{Q}_t}{\mathbf{Q}_t^2(\mathbf{Q}_t-\mathbf{p}_{t,1})^2(\mathbf{Q}_t+\mathbf{p}_{t,2})^2}\times \\ f_g(x_1,x_1',Q_1^2,\mu^2)F(t_1)f_g(x_2,x_2',Q_2^2,\mu^2)F(t_2)\langle\mathcal{M}_{gg\to X}\rangle \tag{5.67}$$

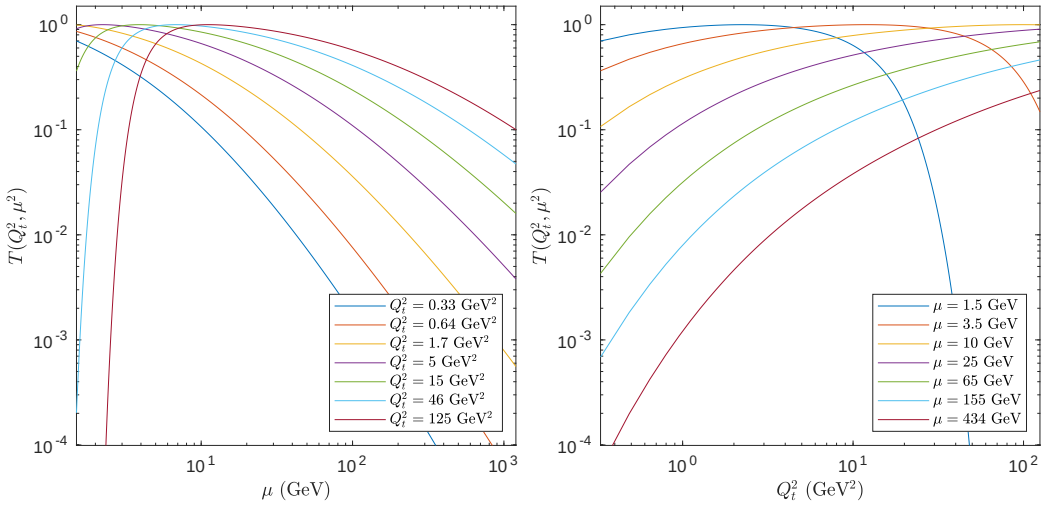

Figure 5.12: The Sudakov suppression factor $T(Q^2, \mu^2)$ evolution.

where the π^2 factor comes from the parton to proton replacement $\alpha_s C_F/\pi \to f_g$. Here we simply assume that the *coherent* proton form factors factorize with the gluon fluxes. The fusing gluon vectors in the loop are $\mathbf{q}_1 = \mathbf{Q}_t - \mathbf{p}_{t,1}$ and $\mathbf{q}_2 = \mathbf{Q}_t + \mathbf{p}_{t,2}$, with the outgoing proton transverse momentum $\mathbf{p}_{t,1,2}$. Here, one sees that the process cross section has a quadratic dependence on the gluon pdfs. This 2D-loop integral is calculated numerically event-by-event and the default scales are chosen as $\mu^2 = M_X^2$ and $Q_i^2 = \min(|\mathbf{Q}_t|^2, |\mathbf{q}_i|^2)$, but these can be varied easily with the program input.

For the sub-amplitude, it is useful to use the helicity basis and use the decomposition [126] of $q_1^i q_2^j \mathcal{M}_{ij}$ which gives

$$J_z^P = 0^+ \quad : -\frac{1}{2}(\mathbf{q}_1 \cdot \mathbf{q}_2)\left[\mathcal{M}_{++} + \mathcal{M}_{--}\right] \tag{5.68}$$

$$J_z^P = 0^- \quad : -\frac{i}{2}(\mathbf{q}_1 \times \mathbf{q}_2)\left[\mathcal{M}_{++} - \mathcal{M}_{--}\right] \tag{5.69}$$

$$J_z^P = +2^+ : +\frac{1}{2}\left([\mathbf{q}_1 \ominus \mathbf{q}_2] + i[\mathbf{q}_1 \oplus \mathbf{q}_2]\right)\mathcal{M}_{-+} \tag{5.70}$$

$$J_z^P = -2^+ : +\frac{1}{2}\left([\mathbf{q}_1 \ominus \mathbf{q}_2] - i[\mathbf{q}_1 \oplus \mathbf{q}_2]\right)\mathcal{M}_{+-} \tag{5.71}$$

with $\mathbf{q}_1 \oplus \mathbf{q}_2 \equiv q_{1,x}q_{2,y} + q_{1,y}q_{2,x}$ and $\mathbf{q}_1 \ominus \mathbf{q}_2 \equiv q_{1,x}q_{2,x} - q_{1,y}q_{2,y}$. For each outgoing helicity combination, let us point out that the sum over incoming helicity states is here *coherent*. The current first implementation includes $gg \to gg$ continuum,

$gg \to \chi_{c0}$ resonance and $gg \to m\bar{m}$ pseudoscalar meson pair amplitudes. The gluon pdfs and running couplings are provided by the LHAPDF6 library [119]. We encourage users to implement their own processes and pay attention to the normalization conventions. The phase space normalization factors are chosen such that the process is compatible with $2 \to N$ kinematics. Interfacing with automated matrix element generators should be the target for the future. For more processes readily available within Durham model we refer the reader to SuperChic 3 MC [90].

Tensor pomeron model

The tensor pomeron model [129] implementation includes central exclusive processes of a two body continuum production of pseudoscalar pairs, vector meson pairs and baryon pairs. The resonance processes implemented here include the production of scalar resonances f_0, pseudoscalar resonances η, η', vector mesons $\rho^0(770)$ and $\varphi(1020)$ via photoproduction and f_2 tensor mesons interfering with the continuum at the amplitude level. The model takes the ansatz that the pomeron should carry a definite Lorentz structure, namely a rank-2 graviton like current, coupling thus in a symmetric way between matter and antimatter. In this picture, it is the vector odderon which provides the anti-symmetric coupling.

A tensor like pomeron has been also recently studied in a holographic AdS/QCD context in [130] constructing a duality between the triple-graviton vertex and the double pomeron fusion production of glueballs. In the classic Gribov Regge theory, the pomeron carries 'sliding spin' as an infinite sum and, depending on the interaction, may or may not coincide with definite Lorentz structures.

We consider the model implemented here as a practical one due to its explicit enough computational rules, however, see also the aesthetics behind other descriptions. An interesting problem is to see, how uniquely can the upcoming LHC or RHIC measurements with forward protons constrain these structures. Based on this and ansatz structures for couplings, one can write down a diverse set of interactions with mesons and baryons using the corresponding Feynman rules [129]. As an example, the tensor pomeron propagator ansatz is

$$i\Delta_{\mu\nu,\kappa\lambda}(s,t) = \frac{1}{4s}(g_{\mu\kappa}g_{\nu\lambda} + g_{\mu\lambda}g_{\nu\kappa} - \frac{1}{2}g_{\mu\nu}g_{\kappa\lambda})(-is\alpha'_P)^{\alpha_P(t)-1}, \tag{5.72}$$

obeying permutation symmetries and identities

$$\Delta_{\mu\nu,\kappa\lambda}(s,t) = \Delta_{\mu\nu,\lambda\kappa}(s,t) = \Delta_{\nu\mu,\kappa\lambda}(s,t) = \Delta_{\kappa\lambda,\mu\nu}(s,t) \tag{5.73}$$

$$g^{\kappa\lambda}\Delta_{\mu\nu,\kappa\lambda}(s,t) = 0, \quad g^{\mu\nu}\Delta_{\mu\nu,\kappa\lambda}(s,t) = 0. \tag{5.74}$$

The tensorial coupling with the proton or antiproton is

$$i\Gamma_{\mu\nu}(p',p) = -i3\beta F_1(t)\left[\frac{1}{2}[\gamma_\mu(p'+p)_\nu + \gamma_\nu(p'+p)_\mu] - \frac{1}{4}g_{\mu\nu}(\not{p}' + \not{p})\right], \tag{5.75}$$

where $t = (p-p')^2$ and $\beta = 1.87$ GeV^{-1}. We may illustrate the central vertex by the double pomeron production of a pseudoscalar resonance. It involves two amplitude structures

$$\begin{aligned} i\Gamma^{(1)}_{\mu\nu,\kappa\lambda}(q_1,q_2) &= i(g_{\mu\kappa}\epsilon_{\nu\lambda\rho\sigma} + g_{\nu\kappa}\epsilon_{\mu\lambda\rho\sigma} + g_{\mu\lambda}\epsilon_{\nu\kappa\rho\sigma} + g_{\nu\lambda}\epsilon_{\mu\kappa\rho\sigma})\times \\ &\quad (q_1-q_2)^\rho(q_1+q_2)^\sigma \qquad (5.76) \\ i\Gamma^{(2)}_{\mu\nu,\kappa\lambda}(q_1,q_2) &= i\{\epsilon_{\nu\lambda\rho\sigma}[q_{1\kappa}q_{2\mu} - q_1\cdot q_2 q_{\mu\kappa}] + \epsilon_{\mu\lambda\rho\sigma}[q_{1\kappa}q_{2\nu} - q_1\cdot q_2 q_{\nu\kappa}] + \\ &\quad \epsilon_{\nu\kappa\rho\sigma}[q_{1\lambda}q_{2\mu} - q_1\cdot q_2 q_{\mu\lambda}] + \epsilon_{\mu\kappa\rho\sigma}[q_{1\lambda}q_{2\nu} - q_1\cdot q_2 q_{\nu\lambda}]\}\times \\ &\quad (q_1-q_2)^\rho(q_1+q_2)^\sigma, \qquad (5.77) \end{aligned}$$

which correspond to ls-couplings $(1,1)$ and $(3,3)$, respectively [131]. In addition, one adds couplings and form factors in the vertex. For the rest of somewhat lengthy building blocks, we refer to the original papers [129, 131], or directly to our code written in high level C++, which follows closely the algebraic notation. The effective decay couplings of resonances, when possible, are computed according to the model spin structure and PDG branching ratios, which in the scalar case matches directly Eq. 5.39.

In the code, we have implemented the Dirac and Lorentz algebra including gamma matrices, Dirac spinors, massive and massless spin-1 polarization vectors and spin-2 tensors, photon, pomeron and fermion propagators and resonance coupling tensors and evaluate the scattering amplitudes helicity combination by combination with explicit component representations. Continuum and resonance amplitudes are summed together at the amplitude level. Computationally speaking, the most complex sub-amplitudes are the rank-6 Lorentz structures for $J = 2$ resonances. Some of the resulting observables are shown in Figure 5.13. All basic identities are implemented as code unit tests, such as completeness relations and normalizations. Performance wise, this slightly 'brute force' approach could be improved in the future by a more streamlined spinor-helicity formalism. The Lorentz index contractions are accelerated with the FTensor tensor algebra C++ expression template library [132] originally designed for numerical general relativity.

We may discuss a bit the Lagrangian structures. As an example: the negative parity of η-meson production is 'compensated', to conserve parity, by the anti-symmetric

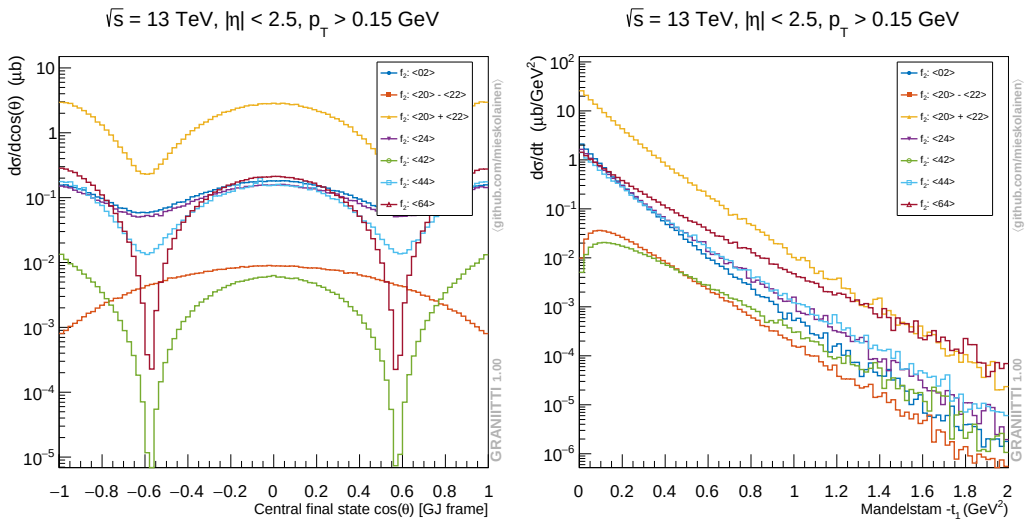

Figure 5.13: PPf_2-vertex rank-6 tensor structures and the resulting angular distributions of final state pions in the system Gottfried-Jackson rest frame and Mandelstam $t_{1(2)}$ distributions. A constant $\langle S^2 \rangle \equiv 0.15$ applied.

epsilon tensor structure which produces characteristic $\sin^2 \Delta\varphi$ cross section dependence motivated experimentally by WA102 data – a well known example. We can take an analogous case from the two photon decays of pseudoscalar and scalar mesons or Higgs. At the effective Lagrangian level, for a pseudoscalar this is a term like $F^{\mu\nu}F^{\kappa\rho}\epsilon_{\mu\nu\kappa\rho}\eta$ whereas for a scalar it would be $F_{\mu\nu}F^{\mu\nu}f_0$, where η, f_0 are the scalar fields and $F^{\mu\nu}$ is the photon vector field. The corresponding amplitudes behave like $e^{\mu\nu\kappa\rho}\epsilon^1_\mu q^1_\nu \epsilon^2_\kappa q^2_\rho$ and $(\epsilon^1_\mu q^1_\nu - \epsilon^1_\nu q^1_\mu)(\epsilon^{2\mu} q^{2\nu} - \epsilon^{2\nu} q^{2\mu})$. Scalar versus pseudoscalar cases can be discriminated if photons are virtual and decay to e^+e^-, a classic example of the π^0 parity determination. This involves analyzing the angular distributions of fermions, which then yields the conclusion that the virtual photon polarizations were orthogonal. Thus, here the forward protons work in an analogous role with different central resonances, which then bootstrapped together with the central system decay products, yield the maximal information about the scattering dynamics and the spin of pomeron. Thus, the full information is in the multidimensional angular distributions.

The main open problems regarding this model parameters are related to the resonance-by-resonance couplings, which require full spectrum simulation compar-

isons with data simultaneously in several differential observables, which is possible. In principle at the LHC energies, one needs to take into account the screening loop (absorption) effects, which is also supported by our code. We return in the discussion to some of the properties of the model.

Jacob and Wick helicity amplitudes

To be able to generate maximally model-independent two-body angular distributions for arbitrary resonances and daughter spin-parities, we have encoded in the relativistic 'wave function free' Jacob and Wick helicity amplitude formalism [133]. Wave function free means that it involves no spinors, polarization vectors and so on. The formalism is abstract, it does not consider the underlying dynamic microscopic details. Because it is algorithmically very easy to rotate the frame or spin density matrix event-by-event, we provide several different spin quantization z-axis (rest frame) options for the event generation, for the definitions, see Section 5.4. We use the Collins-Soper frame (CS) as the default frame for calculating helicity amplitudes. Using this frame the two basic requirements hold: 1. Parity symmetry is manifest, 2. φ-angle dependence stays flat in the laboratory or CM rest frame, as required by rotational invariance.

Canonical and helicity states

In this formalism, there are two different state representations which are intimately connected. The two particle *canonical states* $|JJ_z ls\rangle$ and the *helicity states* $|JJ_z\lambda_1\lambda_2\rangle$ are constructed from the single particle canonical states $|\vec{p}, jm\rangle_i$ and helicity states $|\vec{p}, j\lambda\rangle_i$ for $i = 1, 2$, respectively. Their relation is

$$|\vec{p}, j\lambda\rangle = D^{(j)}_{m\lambda}(\theta_R, \varphi_R)|\vec{p}, jm\rangle, \tag{5.78}$$

where the helicity rotation is defined by (θ_R, φ_R) in terms of Wigner D function. The explicit constructive definitions can be found in [134, 135]. The two particle helicity state is

$$\begin{aligned} &|JJ_z\lambda_1\lambda_2\rangle \\ &\quad = \mathcal{N}_J \sum_{m_1 m_2} \int d\Omega D^{*(J)}_{J_z\lambda}(\theta,\varphi) D^{(s_1)}_{m_1\lambda_1}(\theta,\varphi) D^{(s_2)}_{m_2-\lambda_2}(\theta,\varphi)|\theta\varphi\, m_1 m_2\rangle \qquad (5.79) \\ &\quad = \mathcal{N}_J \int d\Omega D^{*(J)}_{J_z\lambda}(\theta,\varphi)|\theta\varphi\,\lambda_1\lambda_2\rangle, \qquad (5.80) \end{aligned}$$

where $\Omega = (\cos\theta, \varphi)$ denotes the direction of the particle 1 and the normalization which gives simple unit completeness relations is

$$\mathcal{N}_J = \left(\frac{2J+1}{4\pi}\right)^{1/2}. \tag{5.81}$$

Now the explicit relation between the two particle states in the canonical and helicity basis are [135]

$$|JJ_z\lambda_1\lambda_2\rangle = \sum_{ls} \langle\langle l, s, \lambda_1, \lambda_2 | J, s_1, s_2\rangle\rangle \, |JJ_z ls\rangle \tag{5.82}$$

$$|JJ_z ls\rangle = \sum_{\lambda_1\lambda_2} \langle\langle l, s, \lambda_1, \lambda_2 | J, s_1, s_2\rangle\rangle \, |JJ_z\lambda_1\lambda_2\rangle, \tag{5.83}$$

where the re-coupling coefficient double products are denoted with

$$\langle\langle l, s, \lambda_1, \lambda_2 | J, s_1, s_2\rangle\rangle \equiv \mathcal{N}_J^l \langle J\lambda | ls0\lambda\rangle \langle s\lambda | s_1 s_2 \lambda_1, -\lambda_2\rangle, \quad \boxed{\lambda = \lambda_1 - \lambda_2} \tag{5.84}$$

where minus sign between λ_1 and λ_2 comes naturally because individual helicities contribute in opposite directions. The normalization is

$$\mathcal{N}_J^l = \left(\frac{2l+1}{2J+1}\right)^{1/2}. \tag{5.85}$$

The two inner products denote Clebsch-Gordan $SU(2)$ decomposition coefficients, which we evaluate algorithmically via Wigner $3j$ symbols via Racah formula taking into account also the algebraic selection rules. The re-coupling coefficients simply connect the two basis states together

$$\langle J' J'_z ls | JJ_z\lambda_1\lambda_2\rangle = \langle\langle l, s, \lambda_1, \lambda_2 | J, s_1, s_2\rangle\rangle \delta_{JJ'}\delta_{J_z J'_z}. \tag{5.86}$$

To make the notation clear, we have the following variables for the process with spins $J \to s_1 + s_2$:

$$\text{Angular momentum projection}: \; -J \le J_z \le J \text{ with } \mathbf{J} \equiv \mathbf{l} + \mathbf{s} \tag{5.87}$$

$$\text{Helicity}: \; \lambda_J \equiv \mathbf{J}\cdot\mathbf{p}/|\mathbf{p}| \tag{5.88}$$

$$\text{Daughter helicities}: \; -s_1 \le \lambda_1 \le s_1, \; -s_2 \le \lambda_2 \le s_2. \tag{5.89}$$

Note that in our notation $J_z \equiv M$, which we use to emphasize the physical meaning. In the helicity frame, one has $J_z = \lambda_J$. We emphasize that the quantum numbers l

and s are rotational invariants of the canonical basis, just like helicities are rotational invariants of the helicity basis. They are defined by equations for the total spin and fixed orbital angular momentum [135]

$$|\theta\varphi\, sm_s\rangle = \sum_{m_1 m_2} \langle sm_s|s_1 m_1 s_2 m_2\rangle |\theta\varphi\, m_1 m_2\rangle \tag{5.90}$$

$$|lm\, sm_s\rangle = \int d\Omega\, Y_l^m(\theta,\phi)|\theta\varphi\, sm_s\rangle, \tag{5.91}$$

which makes the relations clear.

Density matrix

The resonance state spin polarization is encoded in a fixed spin density matrix ρ_i, which is a $(2J+1)$ hermitian matrix obeying the standard von Neumann density matrix properties and expectation values of operators $\langle A\rangle_\rho = \mathrm{Tr}[\rho A]$, where A can be a spin operator such one of the Pauli matrices for the spin-1/2 case or their generalization. The following properties of the density matrix hold always: A. $\mathrm{Tr}\,\rho = 1$, B. $\rho_{ij}^* = \rho_{ji}$, C. $\rho_{ii} \geq 0$, D. $\mathrm{Tr}\,\rho^2 \leq 1$, E. Positive semi-definite $\leftrightarrow$ non-negative eigenvalues. In general, the matrix can be described by using $(2J+1)^2 - 1$ real parameters. If the density matrix is very complicated, more convenient ways have been developed, see [134].

For a statistical mixture of *pure states*, giving only diagonal entries, the density matrix is

$$\rho = \sum_i p_i |J, J_z\rangle_i \langle J, J_z|_i \quad \text{s.t.} \quad \sum_i p_i = 1. \tag{5.92}$$

As an example a transverse only polarization for $J = 1$ is

$$\rho = \frac{1}{2}|1,-1\rangle\langle 1,-1| + \frac{1}{2}|1,1\rangle\langle 1,1| = \begin{bmatrix} 1/2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1/2 \end{bmatrix}. \tag{5.93}$$

Thus, the representation is

$$|1,-1\rangle \equiv [1,0,0]^T, \quad |1,0\rangle \equiv [0,1,0]^T, \quad |1,1\rangle \equiv [0,0,1]^T, \tag{5.94}$$

which is easily continued for higher spins. Off-diagonal elements in the density matrix are constrained by hermiticity and parity, and they generate the φ-angle dependence. In general, the off-diagonal elements are responsible for the quantum superposition (coherence). A user can freely parametrize the matrices or generate

completely random ones from Gaussian random matrix ensembles. Clearly, the spin density matrix is not covariant but its elements depend on the chosen Lorentz rest frame. An *Unpolarized* process has an equal probability for every helicity state, that is, a diagonal density matrix with elements $1/(2J+1)$ which results in a uniform angular distribution in $(\cos\theta, \varphi)_{\text{r.f.}}$. In the most general case taking into account the parity conservation and hermiticity, $J = 1$ requires 4 and $J = 2$ requires 12 independent parameters.

A natural frame The most natural Lorentz frame for the given process, which always exists but might be non-trivial to know a priori, is the one which gives the most simple, diagonal spin density structure without azimuthal (off-diagonal) dependence. It can be argued to exist, in a mathematical sense, because the density matrix is a hermitian matrix which can be always diagonalized by a suitable rotation. By reciprocity, a process analyzed in an *unnatural frame* will have spurious off-diagonal dependence and mixing of moments. We believe that the CS frame is probably the most natural one, or very close, for central exclusive processes by analog with the Drell-Yan process.

Amplitudes

The decay dynamics is encoded in the helicity amplitude matrix, which is given by the linear combination [136]

$$T^{(J)}_{\lambda_1\lambda_2} = \sum_{ls} \alpha^{(J)}_{ls} \langle\langle l, s, \lambda_1, \lambda_2 | J, s_1, s_2 \rangle\rangle, \tag{5.95}$$

where $0 \leq s \leq s_1 + s_2$ and $0 \leq l \leq J + s$ with dimensions $(2s_1+1)(2s_2+1)$. This relation can be easily inverted, to obtain the canonical ls-representation coefficients in terms of the helicity amplitudes. The unknown coupling weights are denoted with α_{ls}, which are left as a user input. These are normalized such that

$$\sum_{\lambda_1\lambda_2} |T^{(J)}_{\lambda_1\lambda_2}|^2 = \sum_{ls} |\alpha^{(J)}_{ls}|^2 = 1. \tag{5.96}$$

The algorithm takes the sum over all allowed values of ls given the angular momentum conservation, parity and spin-statistics and gives user a list of the required $\alpha^{(J)}_{ls}$ coupling input. Here we remark that some combinations of quantum numbers have very simple α_{ls} coupling structure giving only $T = 1$, and thus only the spin polarization density matrix is unknown. This is the case for example with a $J^P = 2^+$ state into a pseudoscalar pair.

Now we have all the necessary input and the decay transition amplitude matrix element is given by [136]

$$f_{\lambda_1\lambda_2,J_z}(\theta,\varphi) = \mathcal{D}^{(J)}_{\lambda J_z}(\theta,\varphi) T^{(J)}_{\lambda_1\lambda_2}, \tag{5.97}$$

with dimensions $(2s_1+1)(2s_2+1)\times(2J+1)$. Above, the Wigner spin rotation matrix in the spin space is

$$D^{(J)}_{mm'}(\theta,\varphi) = e^{im'\varphi} d^{(J)}_{mm'}(\theta), \tag{5.98}$$

written using the Wigner small d symbol matrix which we calculate algorithmically. The phase convention is fixed by $e^{im'\varphi}$.

Finally, the decay weight or decay amplitude squared of the event is given by the standard expectation value trace

$$|A|^2 = \mathcal{N}_J \mathrm{Tr}[\rho_f] = \mathcal{N}_J \mathrm{Tr}[f \rho_i f^\dagger], \quad \mathcal{N}_J = 2J+1 \tag{5.99}$$

where the operator product maps the initial state ρ_i spin density matrix to the final state spin density matrix ρ_f, also known as the 'Krauss operator' map in quantum mechanics. With the normalization in use, we get $|A|^2 = 1$ for unpolarized decays. In addition, we can directly also calculate coherent spin correlated decay chains by tensor products. For example, $X \to A \to \{A_1 + A_2\} + B \to \{B_1 + B_2\}$ gives

$$f_{tot} = [f_A \otimes f_B]\, f_X \tag{5.100}$$

which are supported. The individual transition amplitudes f are evaluated in the corresponding rest frames of each decay.

To choose different quantization axes, the rotation along direction (θ_R, φ_R) of the quantization coordinate system is obtained via Wigner rotation matrices

$$|J, J'_z\rangle = \sum_{J_z=-J}^{J} \mathcal{D}^{(J)}_{J_z J'_z}(\theta_R,\varphi_R) |J, J_z\rangle. \tag{5.101}$$

The change of basis for the initial density matrix is obtained via a similarity transform

$$\rho'_i = \mathcal{D}^{\dagger(J)}(\theta_R,\varphi_R) \rho_i \mathcal{D}^{(J)}(\theta_R,\varphi_R), \tag{5.102}$$

which is used to change the reference frame of the spin polarization density. The rotation of the density matrix keeps it eigenvalues unchanged, thus also its von

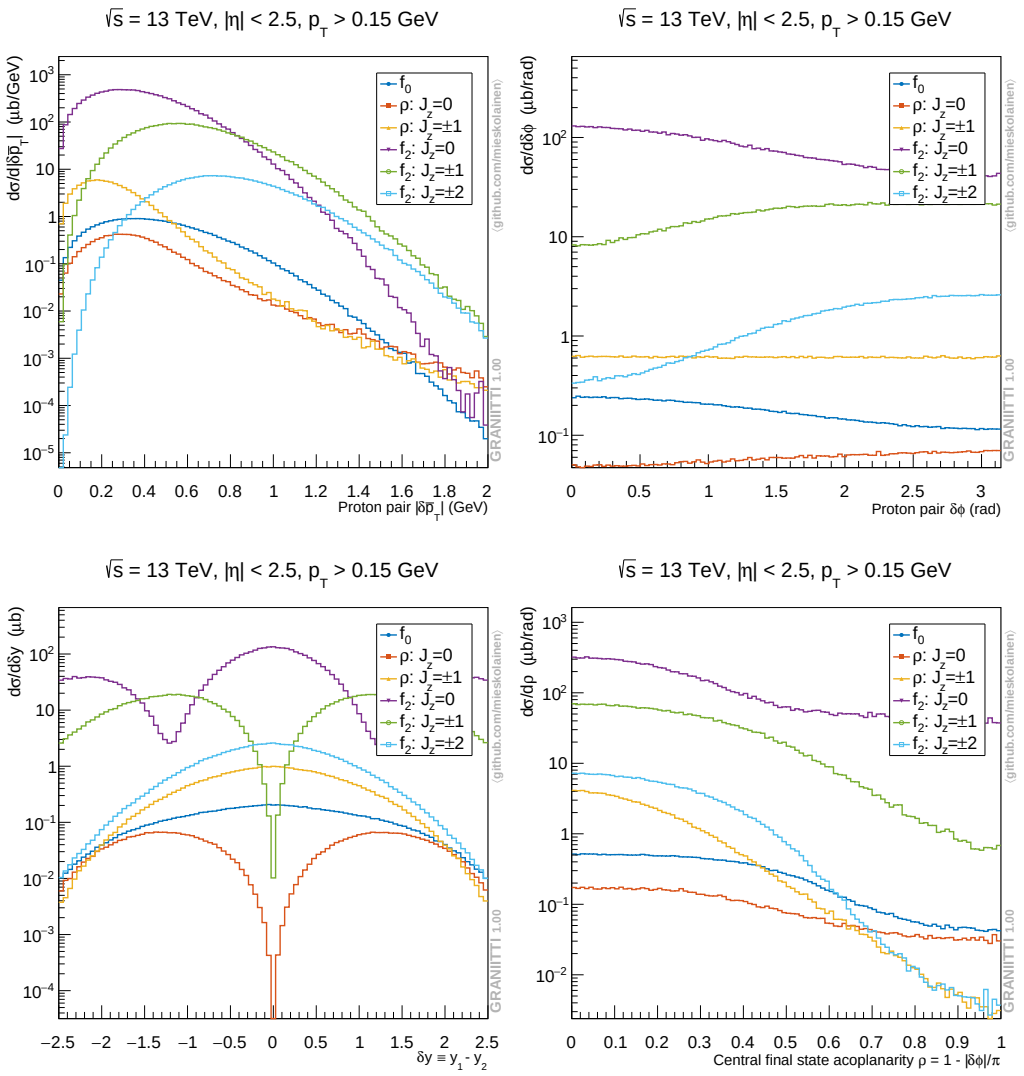

Figure 5.14: The glueball filter forward observable (top left), the forward proton pair transverse angle separation (top right), the central pion pair rapidity separation (bottom left) and the central pion pair acoplanarity (bottom right). A constant $\langle S^2 \rangle \equiv 0.15$ and 0.7 (photoproduction) applied.

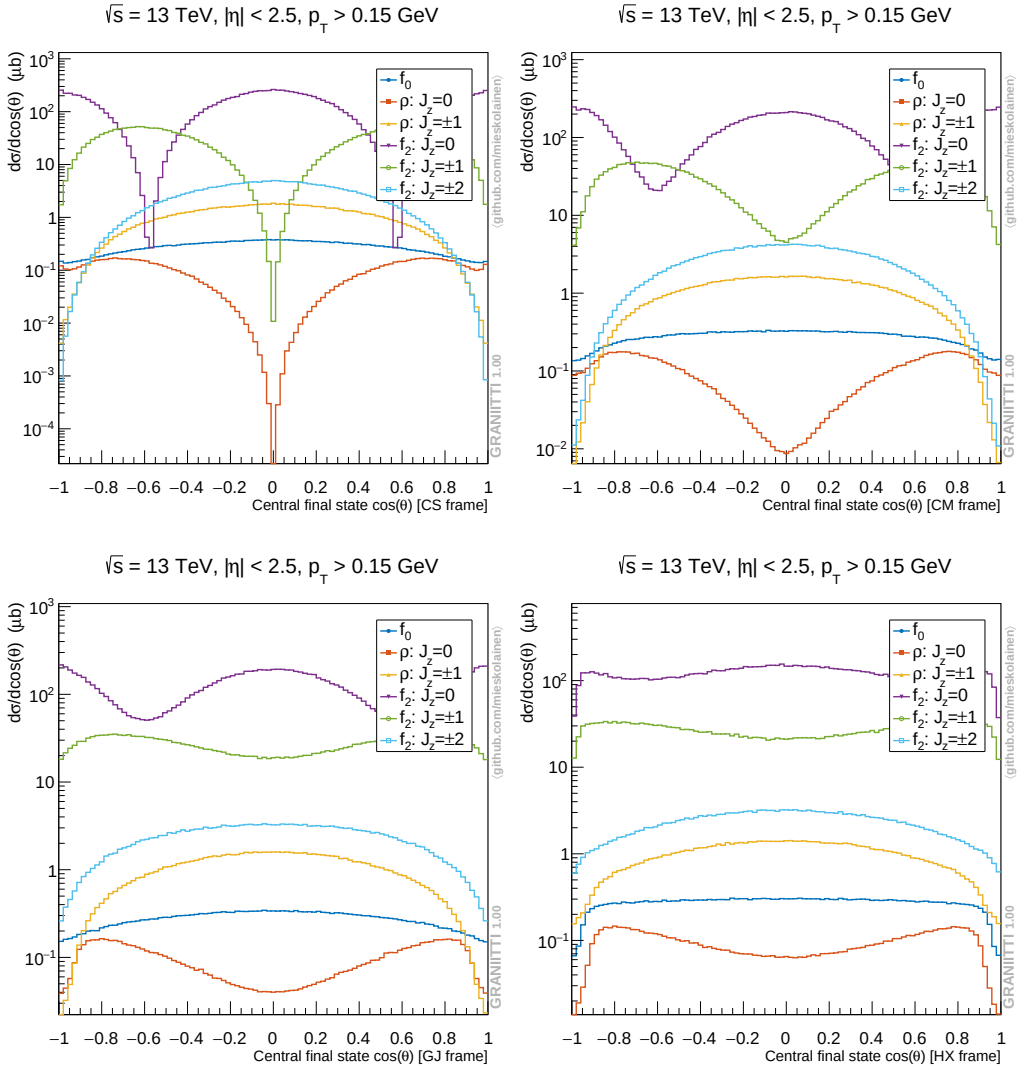

Figure 5.15: Central final state pion $\cos\theta$ measured in different rest frames. A constant $\langle S^2 \rangle \equiv 0.15$ and 0.7 (photoproduction) applied.

Neumann entropy. With parity conservation, the elements of the density matrix obey

$$\rho_{mm'} = (-1)^{m-m'} \rho_{-m-m'}. \tag{5.103}$$

So we need to remember that if we unsuitably rotate the density matrix, the parity conservation may not be manifest anymore, which is clear given that parity is a spatial symmetry.

In Figure 5.14 we demonstrate how the glueball filter $|\Delta\mathbf{p}_t|$, the forward proton transverse angle separation $\Delta\varphi_{pp}$, the central pion rapidity rapidity separation Δy and the central acoplanarity $1-|\Delta\phi|/\pi$ are being driven by the spin polarization of $J=1$ and $J=2$ resonances. We see that the forward observables are strongly correlated with central observables, however, the analysis will be more difficult and ambiguous without forward protons. The $J=2$ states with longitudinal and transverse polarization modes have opposite behavior in terms of forward azimuthal angle, also the glueball filter observable peaks in a different domain. Perhaps the glueball filter should be called a non-perturbative helicity amplitude filter. We can postulate a hypothesis that $J=2$ glueballs may be produced dominantly with $J_z=0$ and quark states with $|J_z|=2$ polarization, but naturally they can have quantum mechanical mixing. In any case, we cannot say that the picture is complete at this point, especially without new data and models working truly at a non-perturbative parton level. Figure 5.15 shows the decay daughter $\cos\theta$ in different rest frames, which demonstrates clearly how different frames smear the distributions due to different rotations. Similarly, different frame rotations will induce non-flat φ-angle dependencies. We also point out that the tensor pomeron model cannot produce $|J_z|=1$ polarization modes for $J=2$ resonances, by angular momentum conservation.

Symmetries

We check algorithmically the required symmetries of the amplitudes to obtain the allowed subset of ls values:

Spin statistics The Bose-Einstein statistics requires $l-s$ to be even for identical boson pairs. The Fermi-Dirac statistics requires $l+s$ to be even for identical fermion pairs. These come simply from the symmetric wavefunction requirement for bosons and anti-symmetric for fermions.

Spatial parity [P] The parity operator or spatial inversion operator $\hat{P}$ with $P|0\rangle = P^{-1}|0\rangle = |0\rangle$, which exchanges a left handed field to a right handed one, operates on the helicity and canonical states as [135]

$$\hat{P}|JJ_z\lambda_1\lambda_2\rangle = P_1P_2(-1)^{J-s_1-s_2}|JJ_z\,-\lambda_1\,-\lambda_2\rangle \tag{5.104}$$

$$\hat{P}|JJ_zls\rangle = P_1P_2(-1)^{l}|JJ_zls\rangle, \tag{5.105}$$

by flipping the sign of helicities λ_1, λ_2, but not the angular momentum projection J_z, because the angular momentum is an axial-vector. Thus, with parity conservation, the helicity amplitudes obey a selection rule [135]

$$T_{\lambda_1\lambda_2} = PP_1P_2(-1)^{J-s_1-s_2}T_{-\lambda_1-\lambda_2}, \tag{5.106}$$

where P, P_1, P_2 are the parity ± 1 of the resonance and daughters. Also, the spherical harmonics are the eigenfunctions of parity

$$\hat{P}Y_l^m(\theta,\varphi) = Y_l^m(\pi-\theta,\varphi+\pi) = (-1)^l Y_l^m(\theta,\varphi). \tag{5.107}$$

Thus, the parity associates with the orbital angular momentum l.

Time reversal [T] The anti-unitary time operator $\hat{T}$ operates to the helicity and canonical states as [135]

$$\hat{T}|JJ_z\lambda_1\lambda_2\rangle = P_1P_2(-1)^{J-J_z}|J\,{-J_z}\lambda_1\lambda_2\rangle \tag{5.108}$$

$$\hat{T}|JJ_zls\rangle = P_1P_2(-1)^{l}|J\,{-J_z}ls\rangle. \tag{5.109}$$

by flipping the sign of J_z but not the helicity of decay daughters.

Charge conjugation [C] The charge parity operator $\hat{C}$ operates by changing the sign of internal quantum numbers. This gives for boson and fermion pairs

$$\hat{C}|\pi^+\pi^-\rangle = (-1)^l(-1)^s|\pi^+\pi^-\rangle = (-1)^{l+s}|\pi^+\pi^-\rangle \tag{5.110}$$

$$\hat{C}|f\bar{f}\rangle = (-1)^l(-1)^{s+1}(-1)|f\bar{f}\rangle = (-1)^{l+s}|f\bar{f}\rangle, \tag{5.111}$$

where the charge conjugation action operates on the orbital part like parity and the third factor in the Fermi case is the particle statistics requirement.

5.3 Kinematics and Monte Carlo sampling

We follow along the lines of the exact 4-body phase space construction suitable for diffraction used in [110, 137], but extend it to include forward proton excitation and generalize it from $N = 4$ process to the case N in two ways: using the exact phase space factorization and a ladder-type direct $2 \to N$ construction.

Skeleton kinematics

The standard QFT cross section in terms of the phase space and amplitude squared is

$$\sigma = \frac{1}{F}\frac{1}{S}\int \prod_{i=1}^{N} \frac{d^3p_i}{(2\pi)^3 2E_i}(2\pi)^4\delta^{(4)}\left(p_A + p_B - \sum_{f=1}^{N} p_f\right)|\mathcal{A}_{2\to N}|^2, \tag{5.112}$$

where the Möller flux is $F = 4\sqrt{(p_A \cdot p_B)^2 + m_A^2 m_B^2} \simeq 2s$ and S is the statistical QFT symmetry factor to take into account identical final states. Now using the relation from Cartesian to collider variables

$$E\frac{d^3\sigma}{d^3p} = \frac{d^3\sigma}{d\varphi dy p_t dp_t} \leftrightarrow \frac{d^3p}{2E} = \frac{1}{2}d\varphi dy p_t dp_t, \tag{5.113}$$

we can turn the sampling over 3-momentum into rapidity, transverse momentum and azimuthal angle. Above, one uses the identity $dy/dp_z = 1/E$.

The total number of Lorentz scalars or non scalar variables needed for $2 \to N$ process is $3N - 4$ for $N \geq 2$. Thus a $2 \to 3$ process needs 5 variables, which we use as our starting point. We can eliminate redundant variables using the energy-momentum conservation. By Lorentz invariance of the expressions, let us work in the frame where $\sum_{i=1}^{3} \vec{p}_i = \vec{0}$ and write

$$\begin{aligned} d\sigma =& \frac{1}{F}\frac{1}{S}(2\pi)^4\delta(E_1 + E_2 + E_3 - \sqrt{s})\times \\ & \delta^{(2)}(\vec{p}_{t,1} + \vec{p}_{t,2} + \vec{p}_{t,3})\delta(p_{z,1} + p_{z,2} + p_{z,3})\prod_{i=1}^{3}\frac{d^3p_i}{(2\pi)^3 2E_i}. \end{aligned} \tag{5.114}$$

1. Eliminate $d^2p_{t,3}$ by $\vec{p}_{t,3} = -(\vec{p}_{t,1} + \vec{p}_{t,2})$ dependence using an implicit integral over the delta function

$$d\sigma = \frac{1}{F}\frac{1}{S}\frac{(2\pi)^4}{(2\pi)^9}\delta(E_1 + E_2 + E_3 - \sqrt{s})\delta(p_{z,1} + p_{z,2} + p_{z,3})\frac{d^3p_1}{2E_1}\frac{d^3p_2}{2E_2}\frac{dp_{z,3}}{2E_3}. \tag{5.115}$$

2. Eliminate $dp_{z,2}$ by $p_{z,2} = -(p_{z,1} + p_{z,3})$ dependence using $d^3p = d^2p_t d_z = d\varphi p_t dp_t d_z$ and an implicit integral over the delta function

$$d\sigma = \frac{1}{F}\frac{1}{S}\frac{1}{(2\pi)^5}\delta(E_1 + E_2 + E_3 - \sqrt{s})\frac{d^3p_1}{2E_1}\frac{d\varphi_2 p_{t,2} dp_{t,2}}{2E_2}\frac{dp_{z,3}}{2E_3}. \tag{5.116}$$

3. Then treat the last energy conservation delta function

$$d\sigma = \frac{1}{F}\frac{1}{S}\frac{1}{(2\pi)^5}\delta(f(p_{z,1}))[d\varphi_1\frac{p_{t,1}}{2E_1}dp_{t,1}dp_{z,1}][d\varphi_2\frac{p_{t,2}}{2E_2}dp_{t,2}][\frac{dp_{z,3}}{2E_3}], \tag{5.117}$$

where we denote

$$f(p_{z,1}) = \sqrt{M_{1,t}^2 + p_{z,1}^2} + \sqrt{M_{2,t}^2 + p_{z,2}^2} + E_3 - \sqrt{s} \tag{5.118}$$

$$p_{z,2}(p_{z,1}) = -(p_{z,1} + p_{z,3}) \tag{5.119}$$

with the transverse mass $M_t^2 \equiv M^2 + p_t^2 = E^2 - p_z^2$. Then, we obtain a factor

$$|\Delta| = \left|\frac{df}{dp_{z,1}}\right| = \left|\frac{p_{z,1}}{\sqrt{M_{t,1}^2 + p_{z,1}^2}} - \frac{p_{z,2}}{\sqrt{M_{t,2}^2 + p_{z,2}^2}}\right| = \left|\frac{p_{z,1}}{E_1} - \frac{p_{z,2}}{E_2}\right|, \tag{5.120}$$

which is in most kinematic cases approximately 2. The need for this is based on the relation

$$\delta(f(x)) = \sum_{x_i : f(x_i)=0} \delta(x - x_i)\left|\frac{df(x_i)}{dx_i}\right|^{-1}, \tag{5.121}$$

where the sum runs over solutions of $f(x) = 0$ (roots). We leave the sum implicit in the notation, because we will use only one root, as we will see.
4. Finally, the change of a variable with $dy_3 = dp_{z,3}/E_3$ gives

$$d\sigma = \frac{1}{F}\frac{1}{S}\frac{1}{(2\pi)^5}|\Delta|^{-1}[d\varphi_1\frac{p_{t,1}}{2E_1}dp_{t,1}][d\varphi_2\frac{p_{t,2}}{2E_2}dp_{t,2}][\frac{1}{2}dy_3]. \tag{5.122}$$

The variables which are thus left to be sampled are the forward system φ_1, φ_2, $p_{t,1}$, $p_{t,2}$ and the central system rapidity y_3. Then, to be able to include variable invariant masses for the forward and central legs, we sample over M_1^2, M_2^2 of the forward systems and over M_3^2 of the central system, the squared masses. The overall Monte Carlo event phase space weight of the main skeleton kinematics is

$$W_{2\to3} = V_{2\to3}\frac{1}{F}\frac{1}{S}\frac{1}{2}\frac{1}{(2\pi)^5}\frac{p_{t,1}}{2E_1}\frac{p_{t,2}}{2E_2}|\Delta|^{-1}. \tag{5.123}$$

The sampling volume is

$$V_{2\to 3} = [p_{t,1}] \times 2\pi \times [p_{t,2}] \times 2\pi \times [y_3] \times [M_3^2], \tag{5.124}$$

where $[x] \equiv |x_{\max} - x_{\min}|$ denotes the sampling interval. In addition, one includes the sampling volumes of M_1^2 and M_2^2, if excitation is included. Also, we need the phase space factor related with the central system phase space, which we will calculate in two ways.

Kinematic polynomials

The variables $p_{z,1}, p_{z,2}, E_1, E_2$ of the event skeleton kinematics are found in a closed form by solving the non-linear system of equations

$$0 = p_{z,1} + p_{z,2} + p_{z,3} \tag{5.125}$$

$$s^{1/2} = E_1 + E_2 + E_3 \tag{5.126}$$

$$E_1^2 = M_1^2 + p_{z,1}^2 + p_{t,1}^2 \tag{5.127}$$

$$E_2^2 = M_2^2 + p_{z,2}^2 + p_{t,2}^2. \tag{5.128}$$

The resulting expressions are very lengthy due to the forward legs and can be found in the code, together with the symbolic machine algebra code solution of the non-linear system, which we used to generate the corresponding C++ code. The polynomial root branch which results in a non-flip of the forward-backward momentum, is chosen.
1. Using the chosen polynomial branch, we calculate

$$p_{z,1} = \mathrm{sol}(M_1, M_2, p_{t,1}, p_{t,2}, p_{z,3}, E_3), \tag{5.129}$$

where the central system energy and longitudinal momentum are

$$\{E_3, p_{z,3}\} = \{M_{t,3}\cosh(y_3), M_{t,3}\sinh(y_3)\}, \tag{5.130}$$

which are obtained in terms of the transverse mass, by solving the central system transverse momentum from the forward system transverse variables, by momentum conservation.
2. Then we get by momentum conservation $p_{z,2} = -(p_{z,1} + p_{z,3})$. The variables E_1 and E_2 are then obtained directly by substitution.

Factorized phase space

To be able to include the central system phase space, we use the exact factorization relation of the phase space

$$d^N\Pi(s; p_1, p_2, \ldots, p_N) \tag{5.131}$$
$$= \frac{1}{2\pi} dM_X^2 \, d^3\Pi(s; p_1, p_2, p_X) d^{N-2}\Pi(M_X^2; p_3, p_4, \ldots, p_N),$$

where p_1, p_2 are the outgoing forward legs, p_X the central system 4-momentum with $M_X^2 \equiv p_X^2$ and $d^N\Pi$ abstracts the corresponding Lorentz invariant phase space measure. The integral over M_X^2 represents the integral over the central system mass squared. The central system flat $1 \to N-2$ phase space is constructed recursively following the classic algorithm of Kopylov and Raubold-Lynch described by James in [138], which calculates also the exact phase space volume weight $W_{1\to N-2}$. The basic idea behind the algorithm is to split the phase space into $N-2$ sequential $1 \to 2$ decays with intermediate masses, for the explicit details of this well known algorithm, we refer reader to the program code. The two body phase space is nearly trivial and thus works as the building block. The total weight of the event is now

$$W_{2\to N} = W_{2\to 3} \frac{1}{2\pi} W_{1\to N-2}. \tag{5.132}$$

The classic algorithm can be plug-in replaced easily with alternative algorithms, such as variants of RAMBO [139]. Also, we have a simple 'chain recursive' phase space implemented which can be useful for long decay chains with intermediate propagators. However, using it requires some care due to intermediate-mass squared sampling. If the matrix element of the process contains all information about the intermediate states, then the sampling should be done with flat masses squared within reasonable ranges, given that the final state leg permutations for different sub-amplitudes may probe different mass regimes in the phase space. Alternatively, a $1 \to N-2$ central phase space can always be constructed, which is safe but with low efficiency if N is large. Finally, if the matrix element does not contain full decay chain information, then a relativistic Breit-Wigner sampling in mass squared is applied as a simple dynamic propagator model.

Direct phase space

As an alternative formulation of the phase space, instead of the factorized phase space, we have constructed a 'direct' $2 \to N$ kinematics based on solving a certain linear system of equations. Let us denote the number of central states with $K =$

$N-2$. For the transverse degrees of freedom, we need $K-1$ transverse momentum k_t variables, $K-1$ azimuthal variables φ. For the longitudinal degrees of freedom, we need K rapidity y variables as our Monte Carlo sampling variables, in addition to the 4 forward system variables as before. Thus, the total number of variables is $3N-4$, which is the minimum possible.

Let us have so-called difference momentum transverse vectors

$$\vec{k}_{t,j} = (k_{t,j}\cos\varphi_j, k_{t,j}\sin\varphi_j), \quad j = 1,\ldots,K-1, \tag{5.133}$$

which we use as the basis of the construction. Let us then write a system of equations

$$\mathbf{b} = A\mathbf{p}, \tag{5.134}$$

where $\mathbf{p}$ denotes the system vector of central final states. This is solved separately for x and y components, which we leave implicit in the notation below. We construct the system vector of difference momentum

$$b_j = \begin{cases} \vec{k}_{t,j} - \vec{w}, & \text{when } \ j = 1 \\ -\vec{k}_{t,j-1} - \vec{w}, & \text{when } \ j = 2,\ldots,K, \end{cases} \tag{5.135}$$

where the forward system transverse vector sum is $\vec{w} = \vec{p}_1 + \vec{p}_2$. Then, we can solve the central final state transverse momentum components by

$$\mathbf{p} = A^{-1}\mathbf{b}. \tag{5.136}$$

The system matrices are full rank with components

$$A_2 = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}, A_3 = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 2 & 1 \\ 1 & 0 & 2 \end{pmatrix}, A_4 = \begin{pmatrix} 2 & 0 & 1 & 1 \\ 0 & 2 & 1 & 1 \\ 1 & 0 & 2 & 1 \\ 1 & 1 & 0 & 2 \end{pmatrix} \ldots, \tag{5.137}$$

which can be constructed with a simple algorithm up to any K and the inverses are taken by symbolic machine algebra and saved. For example

$$A_2^{-1} = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}, A_3^{-1} = \begin{pmatrix} \frac{2}{3} & 0 & -\frac{1}{3} \\ \frac{1}{6} & \frac{1}{2} & -\frac{1}{3} \\ -\frac{1}{3} & 0 & \frac{2}{3} \end{pmatrix}, A_4^{-1} = \begin{pmatrix} \frac{7}{8} & \frac{1}{8} & -\frac{1}{2} & -\frac{1}{4} \\ \frac{3}{8} & \frac{5}{8} & -\frac{1}{2} & -\frac{1}{4} \\ -\frac{1}{8} & \frac{1}{8} & -\frac{1}{2} & -\frac{1}{4} \\ -\frac{5}{8} & -\frac{3}{8} & -\frac{1}{2} & \frac{3}{4} \end{pmatrix}, \tag{5.138}$$

which demonstrate the algebraic structures. Finally, the longitudinal momentum and energy for the central final states are obtained with

$$p_{z,j} = m_{t,j} \sinh y_j \tag{5.139}$$

$$E_j = m_{t,j} \cosh y_j, \quad \text{for} \quad j = 1, \ldots, K, \tag{5.140}$$

where $m_{t,j}^2 = m_j^2 + p_{t,j}^2$. We sample K rapidity variables y_j independently, which then fix the central system rapidity and we can proceed with the skeleton kinematics polynomials.

Then proceeding in a same way as in the $2 \to 3$ case, the total Monte Carlo phase space weight is

$$W_{2\to N} = V_{2\to N} \frac{1}{F} \frac{1}{S} \frac{1}{2^{2(N-2)}} \frac{1}{(2\pi)^{3N-4}} \frac{p_{t,1}}{2E_1} \frac{p_{t,2}}{2E_2} |\Delta|^{-1} \prod_{j=1}^{K-1} k_{t,j}, \quad \text{for} \quad N \geq 4, \tag{5.141}$$

where the sampling volume is

$$V_{2\to N} = [p_{t,1}] \times 2\pi \times [p_{t,2}] \times 2\pi \times [y_j]^K \times ([k_{t,j}] \times 2\pi)^{K-1}. \tag{5.142}$$

Again, if the forward excitation is included, the sampling volumes of M_1^2 and M_2^2 are inserted. Let us point out that we have found this space construction to be easily unstable with VEGAS with high leg count $N \geq 4$ Regge like amplitudes, due to the complicated integration boundaries and a non-trivial structure of the high dimensional Lorentz manifold. Typically, a larger number of integrand evaluations has somewhat stabilized the behavior. In practice we recommend the factorized phase space as the default stable option. However, this phase space construction provides a cross check and may turn out to be of good use with alternative importance sampling techniques. In general, we check all kinematic algorithms against the well known exact volume formulas for the massive two body case and the massless N-body case.

Also, as the simplest possible construction, we include a collinear phase space option $2 \to N$, where N is the number of final states excluding proton remnants. This phase space is suitable for simple amplitudes convoluted with parton densities or collinear EPA fluxes.

Monte Carlo sampling

The basic idea behind Monte Carlo integration with importance sampling in n-dimensional space is

$$I = \int_\Omega d^n\mathbf{x}\, f(\mathbf{x}) = \int_\Omega d^n\mathbf{x}\, \frac{f(\mathbf{x})}{q(\mathbf{x})} q(\mathbf{x}) \simeq \frac{V_\Omega}{N} \sum_{i=1}^{N} \frac{f(\mathbf{x}_i)}{q(\mathbf{x}_i)}, \text{ when } N \to \infty, \tag{5.143}$$

where $V_\Omega = \int_\Omega d^n\mathbf{x}$ is the integration boundary volume, typically a box volume and N is the number of samples. So instead of sampling $\mathbf{x}$ uniformly from $[0,1]^n$, we sample $\mathbf{x}$ according to $q(\mathbf{x})$ and then compensate for this in the integral by weighting events with $1/q(\mathbf{x})$. When $q(\mathbf{x}) \to 0$, no samples are generated, thus there is no division by zero. However, unless $f(\mathbf{x})$ also goes to zero simultaneously, the integral will be biased. Also the normalization

$$\int d\mathbf{x}\, q(\mathbf{x}) = 1 \tag{5.144}$$

needs to hold, which usually needs to be estimated simultaneously. We want to find out the optimal importance sampling pdf

$$q_{\text{opt}}(\mathbf{x}) \equiv \frac{|f(\mathbf{x})|}{\int d^n\mathbf{x}\, |f(\mathbf{x})|}. \tag{5.145}$$

This would give a vanishing variance for the integral estimate, however, the problem of adaptive learning of $q(\mathbf{x})$ is non-trivial.

For sampling the phase space and integrating cross sections, the engine includes a fully multithreaded implementation of classic VEGAS adaptive importance sampling [140], where multithreading is implemented using C++17 standard library threading support by distributing the integrand samples over a fixed number of threads. We have tested the scalability up to thousands of threads. The correctness of VEGAS implementation can be cross-checked with a naive flat sampling mode. In a standard Monte Carlo way, we operate over a unit hypercube $[0,1]^n$ and scale and shift each dimension with $x_i \to a_i + (b_i - a_i)x_i$ to the interval $[a_i, b_i]$. Because describing the correlations in high dimensional phase space is in general highly non-trivial and requires neural networks or similar techniques, VEGAS takes a factorized simplification

$$q(\mathbf{x}) = \prod_{i=1}^{n} q_i(x_i). \tag{5.146}$$

Clearly, neglecting correlations gives bad efficiency if the process kinematics $\times$ matrix element squared does not 'align' along the dimensions. The complexity scales as $\mathcal{O}(nB)$, where B is the number of bins per dimension. In contrast, full n-dimensional histogram representation would scale exponentially fast $\mathcal{O}(B^n)$. Other classic alternatives or extensions are mixture model importance densities, also known as multi-channeling in high energy physics.

VEGAS histogram grids for each dimension need to be initialized over a few number of iterations. The number of iterations R and the number of samples per

iteration N_k in the initialization burn-in phase and in the integration phase are free parameters of the algorithm. In the integration phase, we use also a maximum relative error $\sigma_I/\langle I\rangle$ target as a criterion.

For each k-th iteration, the integral estimate and its variance are

$$\langle I_k\rangle = \frac{1}{N_k}\sum_{i=1}^{N_k}\frac{f(\mathbf{x})}{q^{(k)}(\mathbf{x})},\quad \sigma^2_{I_k} = \frac{1}{N_k-1}\sum_{i=1}^{N_k}\left[\left(\frac{f(\mathbf{x}_i)}{q^{(k)}(\mathbf{x}_i)}\right)^2 - \langle I_k\rangle^2\right], \tag{5.147}$$

where one accumulates the sum of values $f(\mathbf{x})/q(\mathbf{x})$ and their squares during the operation. We evaluate the quality of the set of integral estimates using the χ^2 test

$$\chi^2/\mathrm{ndf} = \frac{1}{N_k-1}\sum_{k=1}^{R}\frac{(\langle I_k\rangle - \langle I\rangle)^2}{\sigma_{I_k}}, \tag{5.148}$$

with values close to unity being an indicator of solid results. Above, the global estimate of the integral is the weighted sum

$$\langle I\rangle = \left[\sum_{k=1}^{R}\frac{N_k}{\sigma_{I_k}}\right]^{-1}\sum_{k=1}^{R}\frac{N_k\langle I_k\rangle}{\sigma_{I_k}}. \tag{5.149}$$

The binning algorithm operates with a fixed number of bins per dimension and shifts the bin boundaries during the operations according to the original description [140]. The stability of the re-binning is controlled with an additional regularization parameter λ. There are also variants of VEGAS, where stratified sampling is combined with importance sampling, but we did not find them effective enough to compensate for the additional complexity. The unweighted event generation is based on a standard hit-and-miss, where estimate for the crucial maximum weight $\max[f(\mathbf{x})/q(\mathbf{x})]$ is obtained during the pre-event generation phase. After the event generation, the user is given the statistics of events overshooting the maximum weight, thus indicating a need for a longer pre-event generation phase.

In addition to VEGAS importance sampling, variables related to steeply falling spectra such as the forward system invariant masses are MC sampled in log space together with the corresponding Jacobian inside the integrand, often with much improved behavior. The rest of the standard kinematics not described here is based typically on heavy use of the Källen triangle function. To point out, we have experimented with deep learning techniques, similar to [141], which could provide superior scaling in higher dimensions and with difficult scattering amplitudes. The results are promising and we may expect these techniques for learning the distribution $q(\mathbf{x})$ to be included in the future versions.

5.4 Analysis engine

The analysis engine includes highly efficient plotting machinery to gain quick understanding of fiducial observables for different processes and theoretical constructions, such as eikonal densities. Naturally, these are also part of the automated test suite to control the quality of the code at a high level.

Lorentz rest frames

There is an infinite number of different Lorentz rest frames for the system X, obtained by fixed or event-by-event kinematics dependent $SO(3)$ rotations. However, certain Lorentz rest frames have more special properties than others. These different frames have been originally designed to be more 'natural' either for s-channel or t-channel dominated processes or mitigate the effects of the system transverse momentum, which is the case with Collins-Soper frame. In practice, it is trivial and highly recommended to repeat the analysis in multiple frames. Different frames give different projections of the angular distributions and spherical moments, by definition.

Let us have beam protons in the lab frame p_1^{lab} and p_2^{lab} and their boosted versions p_1^X, p_2^X in the system X rest frame. We now define a set of frames X', which are related to the frame X by a rotation. The definitions of the z-axis are as follows

$$\text{CM:}\ \ \mathbf{z} = [0,0,1]^T \tag{5.150}$$

$$\text{HX:}\ \ \mathbf{z} = u(-(\mathbf{p}_1^X + \mathbf{p}_2^X)) \tag{5.151}$$

$$\text{CS:}\ \ \mathbf{z} = u(u(\mathbf{p}_1^X) - u(\mathbf{p}_2^X)) \tag{5.152}$$

$$\text{AH:}\ \ \mathbf{z} = u(u(\mathbf{p}_1^X) + u(\mathbf{p}_2^X)) \tag{5.153}$$

$$\text{PG:}\ \ \mathbf{z} = u(\mathbf{p}_1^X)\ \ \text{or}\ \ \mathbf{z} = u(\mathbf{p}_2^X), \tag{5.154}$$

where $u(\mathbf{x}) \equiv \mathbf{x}/\|\mathbf{x}\|$ returns a unit vector.

Center of Momentum (CM) The quantization z-axis in the center of momentum X' is the same as in the X frame. No rotation is involved in the transformation to this frame, only a boost from the colliding proton-proton beam frame (lab).

Helicity (HX) The quantization z-axis is defined by the system X momentum vector direction in the lab frame, or equivalently, by the negative direction of the sum of the initial state proton $\mathbf{p}_1^X$ and $\mathbf{p}_2^X$ momentum in the system X rest frame [142]. This is a common analysis frame in quarkonium studies.

Collins-Soper (CS) The quantization z-axis is defined by the bisector vector between the initial state proton $\mathbf{p}_1^X$ and $-\mathbf{p}_2^X$ (negative) momentum in the system X rest frame. This originated from the context of Drell-Yan process [143], but is not limited to it.

Anti-Helicity (AH) The quantization z-axis is defined by the bisector vector between the initial state $\mathbf{p}_1^X$ and $\mathbf{p}_2^X$ (positive) momentum in the system X rest frame. This frame may be interesting for pure symmetry reasons, because it is perpendicular to the CS frame.

Pseudo-Gottfried-Jackson (PG) The quantization axis defined by the initial state proton $\mathbf{p}_1^X$ (or $\mathbf{p}_2^X$) momentum vector in the system X rest frame [144]. Note that sometimes, this is known directly as the Gottfried-Jackson frame.

For all frames except CM which has $\mathbf{y} = [0, 1, 0]^T$, we define the y-axis as the normal vector from the plane spanned by the initial states

$$\mathbf{y} = u(u(\mathbf{p}_1^X) \times u(\mathbf{p}_2^X)). \tag{5.155}$$

Finally, the x-axis is obtained by taking the cross product

$$\mathbf{x} = \mathbf{y} \times \mathbf{z}, \tag{5.156}$$

with the axes being orthonormal. These give us a rotation matrix

$$R = [\mathbf{x}, \mathbf{y}, \mathbf{z}]^T, \tag{5.157}$$

which we apply to all boosted final states in the system X

$$\mathbf{p}_i^{X'} \leftarrow R\mathbf{p}_i^X \tag{5.158}$$

to transform them to the new frame X'.

Forward proton dependent frames The quantization z-axis may be defined by the momentum transfer vector $q_1^{lab} = p_1^{lab} - p_1'^{lab}$ (or $q_2^{lab} = p_2^{lab} - p_2'^{lab}$) momentum boosted to the system X rest frame, also known as the Gottfried-Jackson (GJ) frame. In the lab frame the production plane is spanned by $\mathbf{q}_1^{lab}$ and $\mathbf{q}_2^{lab}$. In the GJ rest frame the momentum transfer vectors are back-to-back $\mathbf{q}_1^{GJ} = -\mathbf{q}_2^{GJ}$ along the z-axis. The transform to this frame from the lab can be obtained by two rotations after the boost, which fix also the x- and y-axes. We note here that the boosts and rotations do not commute, so the order counts.

Density matrix in terms of spherical tensors

It is often useful to represent the density matrix in terms of generalized spin-J operators, also known as spherical tensors [134]

$$[T_l^m]_{J_z J_z'} \equiv \langle J J_z | \hat{T}_l^m | J J_z' \rangle \equiv \langle J J_z | J J_z'; lm \rangle, \; 0 \leq l \leq 2J, \; -l \leq m \leq l, \tag{5.159}$$

defined in terms of Clebsch-Gordan coefficients. However, we note here that there can be many other representations too, this one being one of the most common. The rank of the tensor operator is denoted with l. Now the matrix valued multipole expansion is

$$\rho = \frac{1}{2J+1} \sum_{l,m} (2l+1) t_l^{*m} \, T_l^m \tag{5.160}$$

with expansion coefficients (multipole parameters)

$$t_l^{*m} = \sum_{J_z, J_z'} \langle J J_z | J J_z'; lm \rangle \rho_{J_z J_z'} \;\; \text{with} \;\; t_{-l}^m = (-1)^m t_l^{*m}. \tag{5.161}$$

These are normalized with $t_0^0 = \mathrm{Tr}\, \rho = 1$, such that the expectation value is $\mathrm{Tr}\,(\rho T_l^m) = t_l^m$ or $\mathrm{Tr}\,(\rho T_l^{\dagger m}) = t_l^{*m}$. Next we describe how to extract the expansion coefficients from the data.

Spherical harmonics inverse expansion

The angular distribution expansion engine is based on a complete spherical harmonics expansion described in [145]. We shall re-derive it, add some extra rigor regarding the phase spaces and inversion algorithms and clarify some aspects relevant at high energies. Experimentally, with forward protons and Mandelstam t_1, t_2 measured, one is interested in general in the multidifferential cross section

$$\frac{d^5 N}{dt_1 dt_2 dM^2 dY d\Omega} \sim \mathcal{I}(\Omega; t_1, t_2, M^2, Y). \tag{5.162}$$

Alternatively, without forward protons and also integrating over typically flat rapidity Y dependence of the process in the central domain, one has

$$\frac{d^3 N}{dM^2 dP_t d\Omega} \sim \mathcal{I}(\Omega; M^2, P_t), \tag{5.163}$$

where $\Omega \equiv (\cos\theta, \varphi)$ is the decay daughter momentum direction in the chosen rest frame of the system X with invariant mass squared M^2 and transverse momentum P_t. For every single crucial kinematic variable which is integrated (summed) over, a Monte Carlo event generator dependent bias is being induced through the acceptance expansion – unless the event generator is one-to-one with reality. This event generator dependence is even higher with naive single dimensional histogram based efficiency corrections, naturally. That is, fully MC generator independent results can be obtained *only* through fully differential hyperbinning, or by fully differential continuum techniques such as our *DeepEfficiency* which is based on inverting the high dimensional efficiency response via Deep Learning [146].

Phase spaces

We need three different spaces: a detector space $\Omega^{\rm det}$, a fiducial phase space $\Omega^{\rm fid}$ and an angular flat phase space $\Omega^{\rm flat}$. Our definition of the detector space contains the reconstructed and selected events *with the fiducial cuts applied*, thus this space includes any finite efficiency effects. Naturally, it includes also possible biases (offset) and variance (resolution) effects of the track momentum measurements, which can be corrected a posteriori after the spherical expansion, if needed, by unfolding the spectrum histograms. We remark here that in general, there can be also measured events outside the fiducial domain, such as events with very low transverse momentum tracks, but still reconstructed. Thus, one needs careful definitions. The pure fiducial phase space contains the geometric η-acceptance of the detector equipped with a minimum p_t-cutoff, basically the geometric 'ideal' of the detector space and minimally extrapolating. The angular flat phase space is the space that leaves scalar decays uniform over the solid angle, for example, a limited range on the system rapidity – any cut applied only on the system will leave scalar decays uniform, by Lorentz invariance. One could take also the full 4π solid angle space as the flat phase space definition, but that usually means massive extrapolation from the detector (or fiducial) phase space at high energy experiments, easily a factor of 10. An example of a practical definition is

$$\text{Angular flat phase space } \Omega^{\rm flat} = \{|Y_X| < 2.5\}. \tag{5.164}$$

$$\text{Fiducial phase space } \Omega^{\rm fid} = \{|\eta| < 2.5 \ \wedge \ p_t > 0.15 \text{ GeV}\} \tag{5.165}$$

$$\text{Detector phase space } \ \Omega^{\rm det} = \Omega^{\rm fid} \circledast \text{ detector response function.} \tag{5.166}$$

That is, the system's rapidity limit is taken the same as the pseudorapidity of the decay daughters, which is fine given that always monotonically $|\eta| \geq |y|$ for single

final states, which bounds the system rapidity. That is, we must have

$$\Omega^{\rm det} \subseteq \Omega^{\rm fid} \subseteq \Omega^{\rm flat} \subseteq \Omega^{4\pi}, \tag{5.167}$$

so that all events in the detector space belong to the fiducial phase space, and all in the fiducial phase space belong to the flat phase space. In Eq. 5.167, the first relation dictates the efficiency corrections, the second relation dictates the amount of geometric-kinematic inversion (extrapolation) done by the spherical harmonics expansion and the last relation is obvious. The ratio $|\Omega^{\rm fid}|/|\Omega^{\rm flat}|$ is order of 0.4 - 0.7 at the LHC for our processes with the type of definition given by Eqs. 5.165 and 5.164. The generator level 'ground truth' sample needs to contain cuts of the flat phase space. Our code includes examples that illustrate this straightforward but crucial detail. The crucial thing to understand is that the spherical moments are mixed in the pure fiducial phase space, unless the detector acceptance is extremely good. This is the ultimate reason to use the angular flat phase as the inversion target, even in principle the fiducial measurements should be always the primary target, because they minimize data extrapolations, by construction.

Spherical harmonics

The 'intensity' function $\mathcal{I}$ describes the angular distribution. This function is expanded in terms of spherical harmonic moment expansion

$$\mathcal{I}(\Omega) = \sum_{l=0}^{l_{\rm max}} \sum_{m=-l}^{l} t_{lm} Y_{lm}(\Omega), \tag{5.168}$$

where we use a *real valued representation* of the Laplace spherical harmonics, which is an alternative to the complex representation. We illustrate these functions in Figure 5.16. Both the real and complex representation provide a full orthonormal basis for square-integrable functions. The conversion from the complex representation is

$$Y_{lm} \equiv {\rm Real}[Y_l^m] \equiv \begin{cases} \sqrt{2}(-1)^m {\rm Im}[Y_l^m] & \text{if } m < 0 \\ Y_l^0 & \text{if } m = 0 \\ \sqrt{2}(-1)^m {\rm Re}[Y_l^m] & \text{if } m > 0. \end{cases} \tag{5.169}$$

The conversion in the inverse direction is

$$Y_l^m \equiv {\rm Complex}[Y_{lm}] \equiv \begin{cases} \frac{1}{\sqrt{2}}(Y_{l|m|} - iY_{l,-|m|}) & \text{if } m < 0 \\ Y_{l0} & \text{if } m = 0 \\ \frac{(-1)^m}{\sqrt{2}}(Y_{l|m|} + iY_{l,-|m|}) & \text{if } m > 0, \end{cases} \tag{5.170}$$

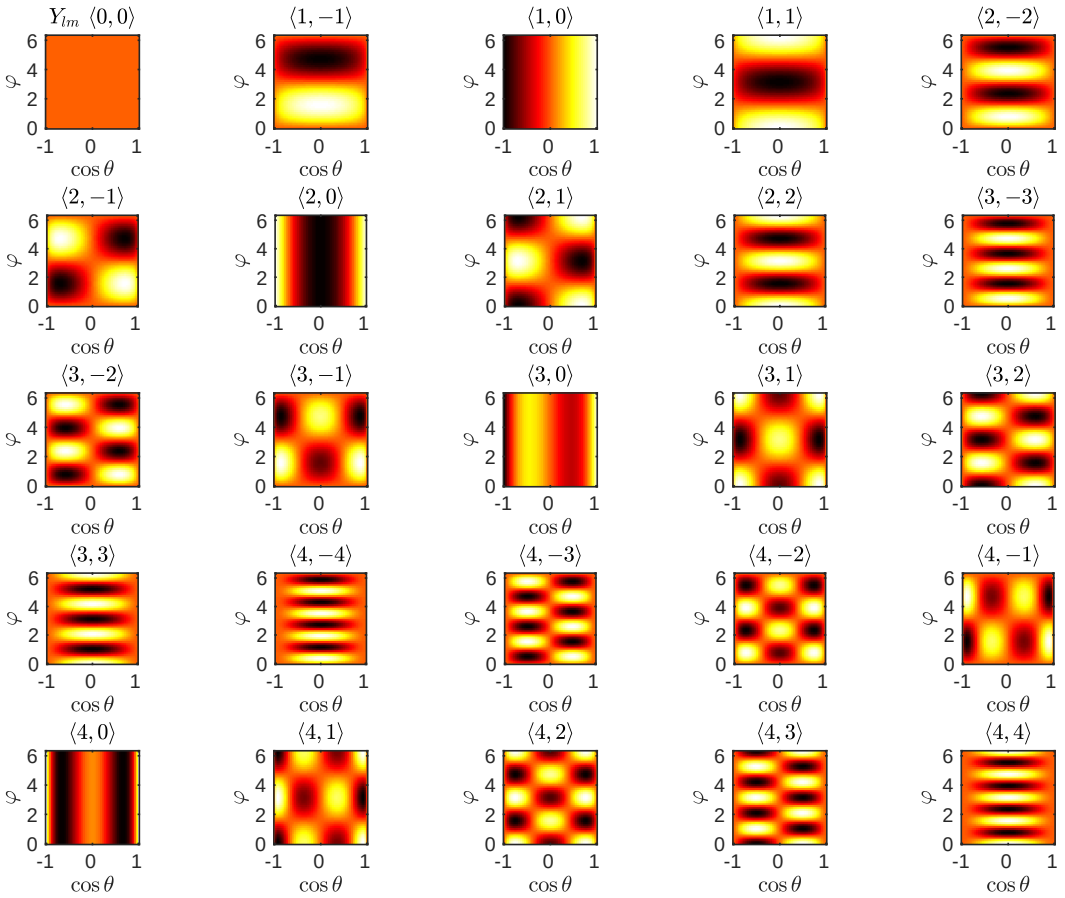

Figure 5.16: Real spherical harmonics Y_{lm} basis functions up to $l_{\max} = 4$.

which obey the basic symmetry relation

$$Y_l^{m*}(\theta, \varphi) = (-1)^m Y_l^{-m}(\theta, \varphi). \tag{5.171}$$

The harmonic functions are calculated using standard numerical methods up to any lm-value using the quantum mechanics normalization conventions.

The expansion coefficients (moments) t_{lm} we are interested in are defined by the inner product integral

$$t_{lm} \equiv \int d\Omega \, \mathcal{I}(\Omega) Y_{lm}(\Omega) \tag{5.172}$$

with normalization

$$t_{00} = \int d\Omega\, \mathcal{I}(\Omega), \tag{5.173}$$

returning the number of events, typically. The inner product works because our basis functions are orthonormal.

Parity inversion $(\theta, \varphi) \mapsto (\pi - \theta, \pi + \varphi)$ gives the relation

$$Y_{lm}(-\mathbf{r}) = (-1)^l Y_{lm}(\mathbf{r}), \tag{5.174}$$

where $\mathbf{r}$ is a unit vector and $(-1)^l \equiv$ parity. With parity conservation in the process and the chosen rest frame, only *even* values of l are needed, because then the values of t_{lm} will integrate to zero for odd l. Now if the processes are also rotation symmetric with respect to φ, we see that it is enough to write the expansion in Eq. 5.168 only over non-negative m. This is seen also in Figure 5.16, where for negative m the basis functions are odd under translations of φ over $0 \mapsto 2\pi$, thus giving a vanishing t_{lm} integral for φ-symmetric distributions. However, if the detector response induces major asymmetries which should be also included in the detector simulation, one may need to use the full expansion.

Inverse expansion

The crucial acceptance $\times$ efficiency function moments are

$$\mathcal{E}_{lm} = \int d\Omega\, A(\Omega) Y_{lm}, \tag{5.175}$$

where the acceptance $\times$ efficiency function $A(\Omega)$ is known only indirectly through detector simulated samples. In its simplest form, it corresponds to geometric fiducial cuts of the detector. The rest of the detector efficiency effects are then variations on this manifold.

The acceptance $\times$ efficiency mixing matrix is

$$\mathcal{E}_{lm,l'm'} = \int d\Omega\, Y_{lm}(\Omega) Y_{l'm'}(\Omega) A(\Omega), \tag{5.176}$$

which describes how the experimental acceptance cuts will 'mix' different spherical moments. In the limit $A(\Omega) \to 1$ we get $\mathcal{E}_{lm,l'm'} = \delta_{ll'}\delta_{mm'}$, by orthonormality. We calculate the Singular Value Decomposition (SVD) based condition number of the matrix $\mathcal{E}$ as

$$\kappa(\mathcal{E}) = \frac{\sigma_{\max}(\mathcal{E})}{\sigma_{\min}(\mathcal{E})}, \tag{5.177}$$

which we use to characterize the geometric ill-posedness of the problem characterizing the limited detector geometric acceptance, but also the simulation Monte Carlo sample size sufficiency given the chosen expansion truncation l_{max}. The maximum and minimum singular values of the matrix are denoted with $\sigma_{\max}, \sigma_{\min}$. The identity matrix has a condition number 1, whereas an ill-posed problem has a very large value of κ.

The *observed* moments of data are

$$t_{lm}^{\rm obs} = \frac{\int d\Omega\, Y_{lm}(\Omega)\mathcal{I}(\Omega)A(\Omega)}{\int d\Omega\, \mathcal{I}(\Omega)A(\Omega)}, \tag{5.178}$$

which are calculated through a finite sum over the hyperbin sample. One needs to pay attention to normalization, that is, we use the standard quantum mechanics normalization of spherical harmonics $\int |Y_l^m|^2 d\Omega = 1$ which conserves probability. To conserve number of events, we multiply each finite sum with $\sqrt{4\pi}$.

The inverse estimate in the flat phase space is given by the direct algebraic inverse

$$\hat{t}_{lm}^{\rm flat} = \sum_{l'm'} [\mathcal{E}_{lm,l'm'}]^{-1} t_{lm}^{\rm obs}, \tag{5.179}$$

which we take through SVD with possible regularization. That is, one does not necessarily need to do Maximum Likelihood optimization. Note here that once we have $\hat{t}_{lm}$ in the flat phase space, we can *push forward* it to the fiducial phase space using the fiducial acceptance map $\mathcal{F}$ which is calculated analogously to the acceptance $\times$ efficiency matrix, leading to

$$\hat{t}_{lm}^{\rm fid} = \sum_{l'm'} \mathcal{F}_{lm,l'm'} \hat{t}_{lm}^{\rm flat}. \tag{5.180}$$

With low event count statistics, usually the most optimal approach to find $\hat{t}_{lm}^{\rm flat}$ is a $d\Omega$-unbinned extended Maximum Likelihood formulation with Poissonian event fluctuations in the hyperbin. The likelihood functional is

$$\mathcal{L} = \frac{\langle n\rangle^n}{n!} e^{-\langle n\rangle} \prod_{i=1}^{n} \frac{\mathcal{I}(\Omega_i)}{\int \mathcal{I}(\Omega)A(\Omega)d\Omega}, \tag{5.181}$$

$$\text{where } \langle n\rangle \equiv \int \mathcal{I} d\Omega\, (\Omega)A(\Omega) = \sum_{lm} \mathcal{E}_{lm} \hat{t}_{lm}^{\rm flat}, \tag{5.182}$$

where the expected number of *observed* events is $\langle n\rangle$, thus the denominator and Poisson term cancel partially. Now taking the negative logarithm for the minimization

gives

$$-\ln\mathcal{L} = -\sum_{i=1}^{n}\ln\sum_{lm}\hat{t}_{lm}^{\rm flat}Y_{lm}(\Omega_i) + \sum_{lm}\mathcal{E}_{lm}\hat{t}_{lm}^{\rm flat}\,, \tag{5.183}$$

where we dropped the constant terms depending only on n. This non-convex optimization problem is minimized numerically via MINUIT routines, more specifically by Davidon-Fletcher-Powel quasi-Newton algorithm (MIGRAD), initial estimate given by the algebraic inverse. Once the set of moments $\{t_{lm}^{\rm flat}\}$ is extracted, one may construct other observables based on this set, such as density matrices. We point out here that for *each hyperbin independently*, one repeats the whole chain of calculations including the detector expansion matrices. Technical extensions of this could include interpolation between hyperbins, to suppress statistical fluctuations of the simulation and data samples.

We demonstrate the spherical harmonics expansion in Figure 5.17, 5.18, 5.19, 5.20, 5.21 and 5.22 with cuts suitable for the ATLAS and CMS experiments, where we have neglected the forward proton cuts, thus also the photoproduced ρ^0 is well visible. We emulated the detector p_t-efficiency transfer function with a smooth hyperbolic tangent function for illustration and used flat η-efficiency within the acceptance cube. The corresponding acceptance decompositions are shown in Figures B.1, B.2 and B.3 in Appendix. The somewhat peculiar acceptance bowl at low masses is due to the interplay between the peripheral kinematics and phase space definitions.

The spectra are normalized to one, but naturally, one may use event counts or cross sections. The figures contain results in both the fiducial phase space and in the inverted flat phase space. The pure $J = 0$ process verifies the correctness of the inversion algorithm giving no visible moments other than $lm = \langle 0, 0\rangle$ in the flat phase space, as should be the case. All other frames than the CM frame have non-zero coefficients with $m \neq 0$. The case $m = 0$ reduces always to the ordinary Legendre polynomials

$$Y_{l0}(\theta,\varphi) = \sqrt{\frac{2l+1}{4\pi}}P_l(\cos\theta), \tag{5.184}$$

with no φ dependence. However, one needs to remember that also in this case the flat phase space inversion machinery is crucial for easy interpretations, otherwise we could just use directly the ordinary Legendre polynomials without any algorithmic machinery.

The reference $J = 0$ process used for the acceptance expansion is on purpose here taken, for the realism, with a slightly different effective b-slope than the full spectrum process. The result is that we obtain slightly varying inversion results in

different frames which is visible especially at low masses. This is just the Monte Carlo model dependence which is propagated differently in different frames. This dependence is minimized by using a b-slope value closely matching the data. If event statistics allow, binning over the system transverse momentum solves the problem.

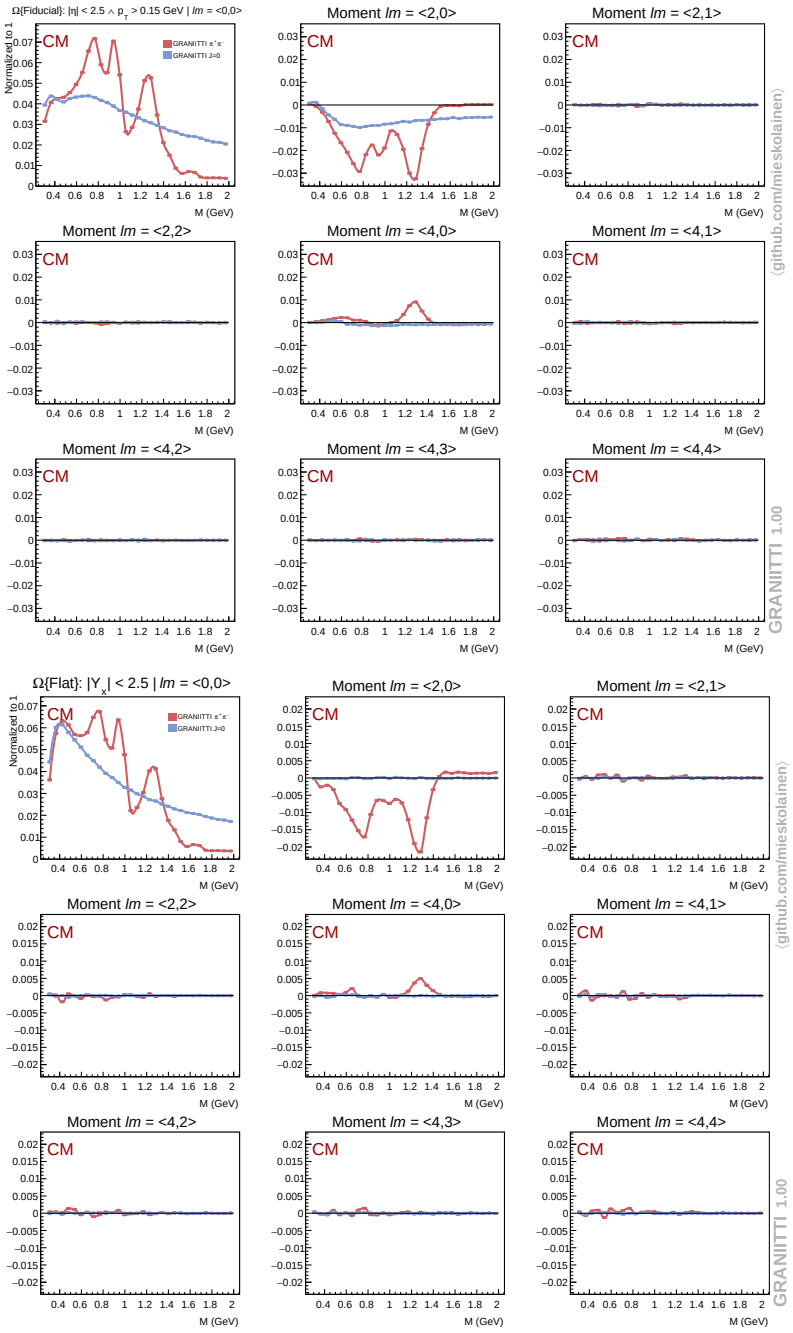

Figure 5.17: CM frame: Harmonic moments in the fiducial phase space (rows 1-3) and in the flat phase space (rows 4-6).

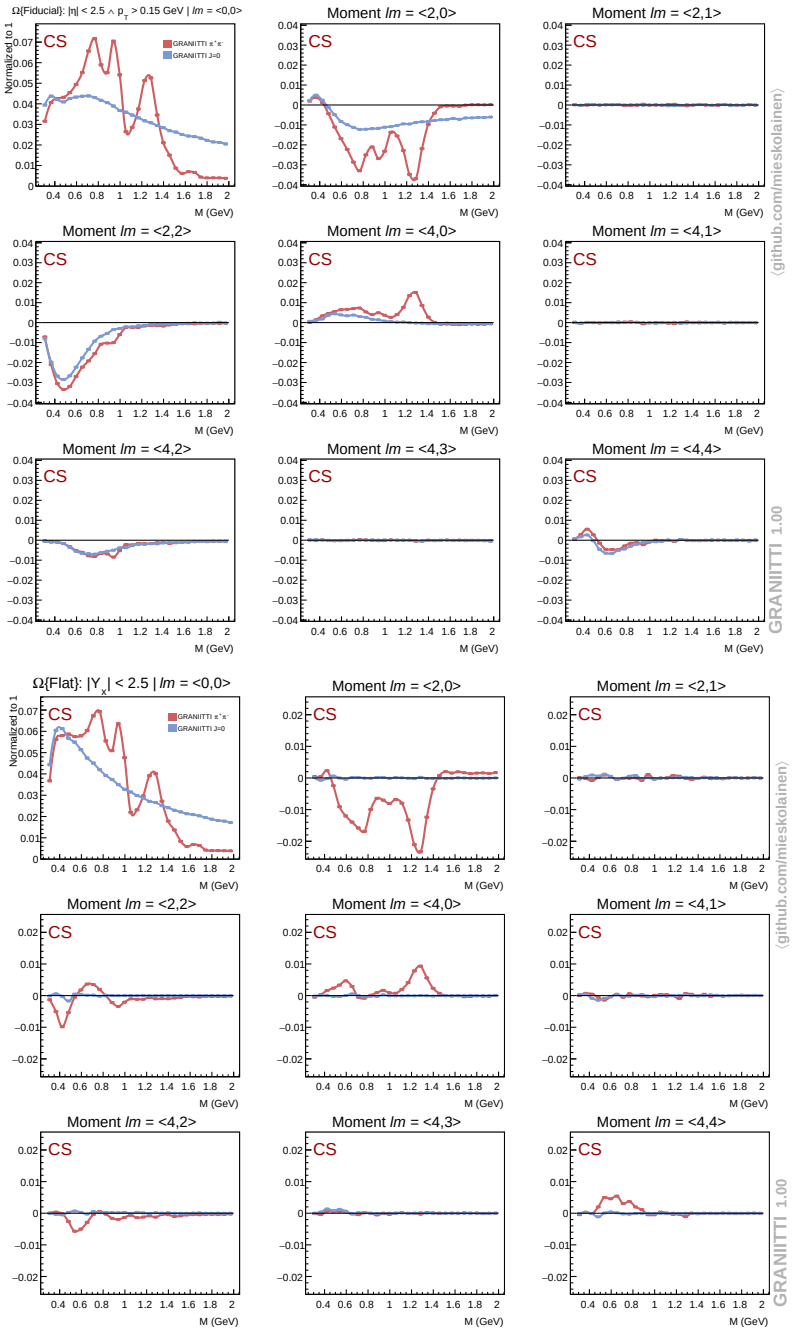

Figure 5.18: CS frame: Harmonic moments in the fiducial phase space (rows 1-3) and in the flat phase space (rows 4-6).

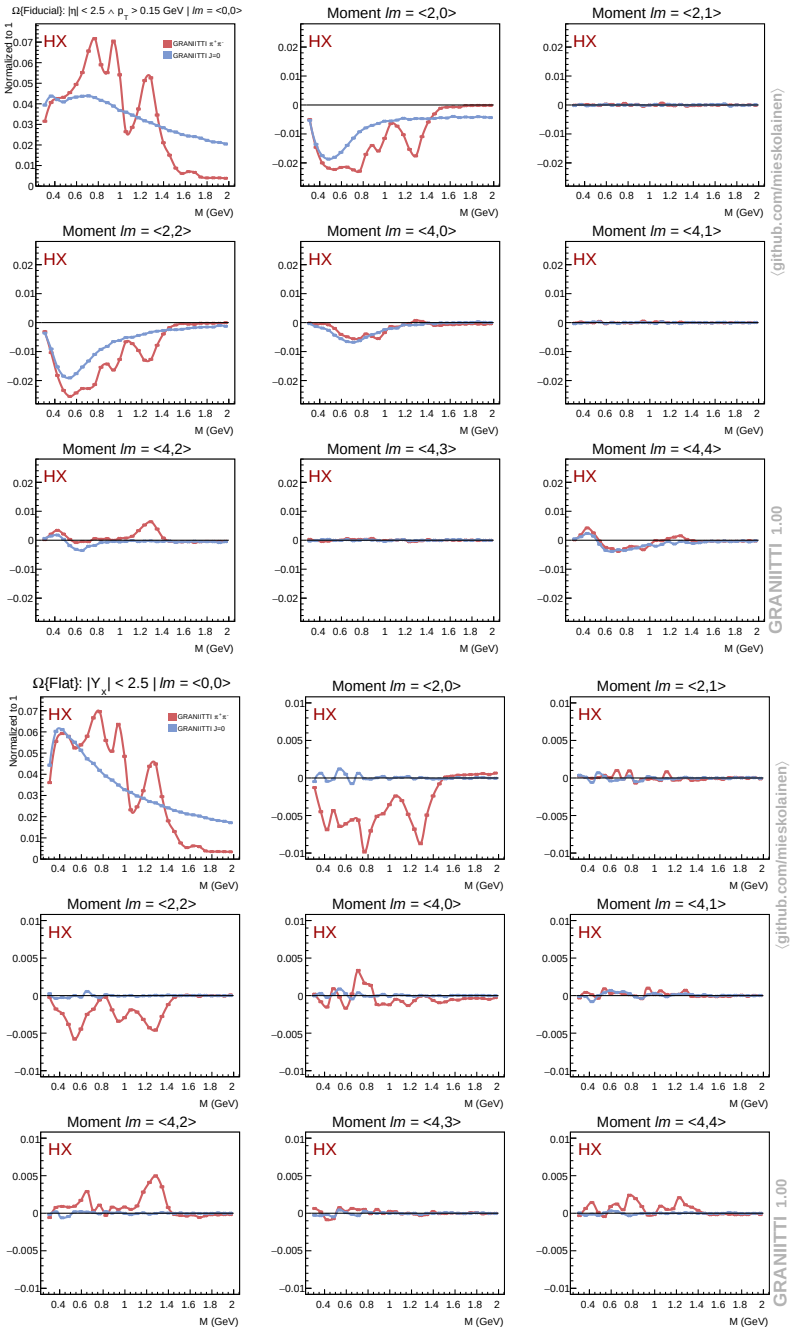

Figure 5.19: HX frame: Harmonic moments in the fiducial phase space (rows 1-3) and in the flat phase space (rows 4-6).

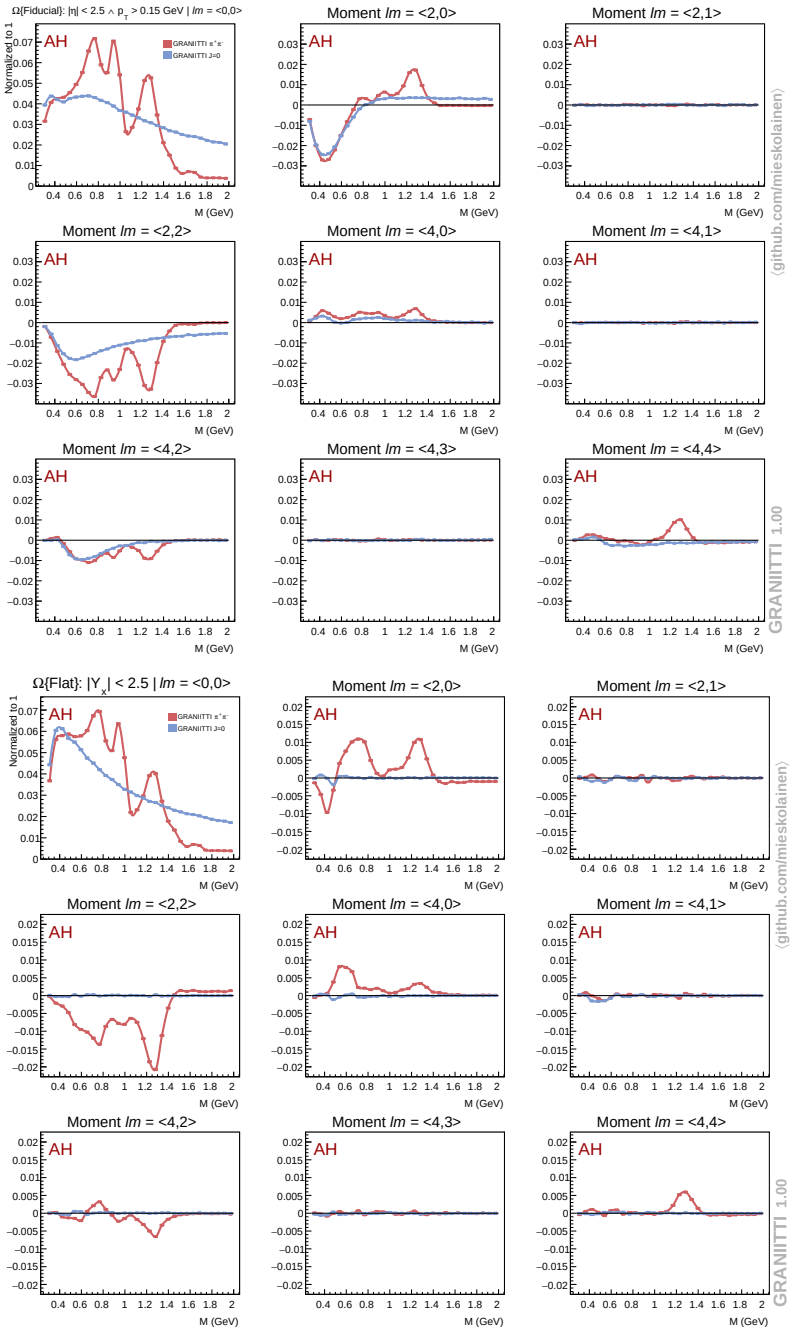

Figure 5.20: AH frame: Harmonic moments in the fiducial phase space (rows 1-3) and in the flat phase space (rows 4-6).

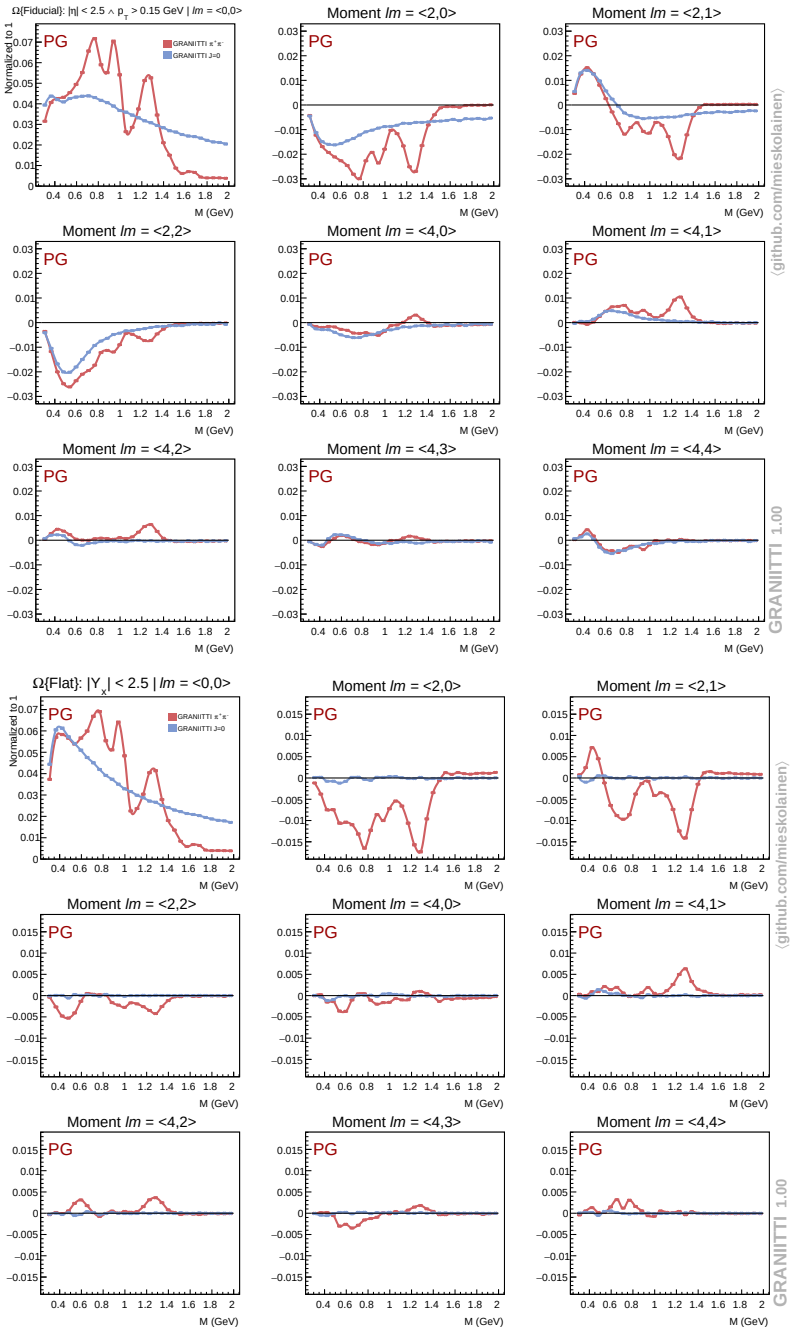

Figure 5.21: PG frame: Harmonic moments in the fiducial phase space (rows 1-3) and in the flat phase space (rows 4-6).

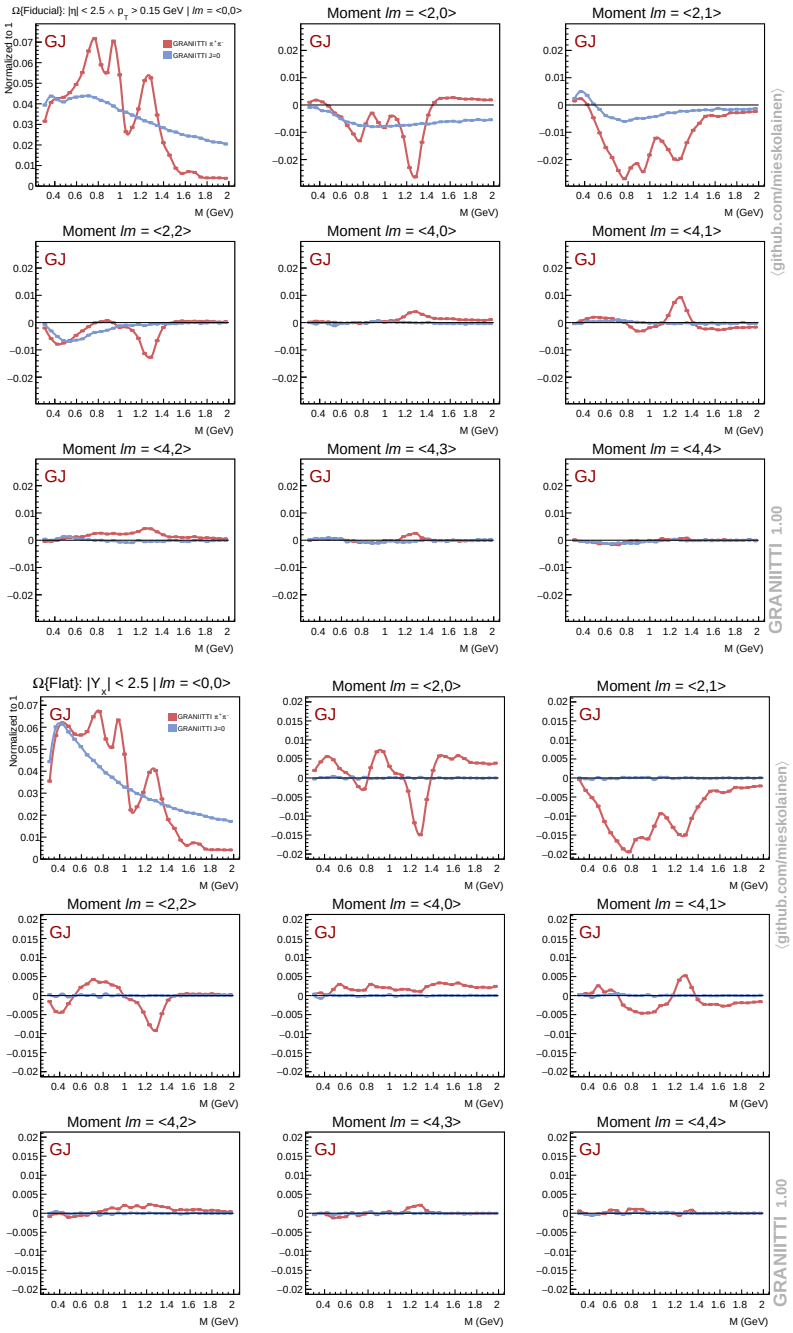

Figure 5.22: GJ frame: Harmonic moments in the fiducial phase space (rows 1-3) and in the flat phase space (rows 4-6).

5.5 Technology

The event generator is implemented in modern C++, utilizing features from the C++17 language standard with full parallel processing using multithreading, basically with no limit in the number of threads. As an example of the current state-of-the-art hardware, Intel Xeon Platinum CPU provides hardware support up to 112 threads with 56 cores. The default random number generator in use is 48-bit RANLUX [147] with easy command line seeding for the distributed grid computing use. The inline commented and physics literature referenced codebase is currently $\mathcal{O}(35\text{k})$ lines.

The generator steering is done using JSON5 style input control cards and a command line interface. To ease out using the software correctly, we check the input and use extensively exception throwing for failure situations. Free parameters of the models are easily changed by modifying the system JSON cards which are grouped under the same folder which allows creating easily different 'tunes'. Event output is provided in HepMC3 (default), HepMC2 and HEPEVT formats and a converter to LHE format is provided. The ROOT 6 library is used for the analysis algorithms, fitting and plotting, but the generator side is a standalone code and should compile on any modern Linux platform. Compiling the source code is fully automated with standard MAKE tools and a GCC7+ or Clang5+ compiler is needed.

The code is constructed with modern quality standards, driven by an extensive set of fully automated and semi-automated custom test cases and Catch2 library, a software development aspect which we have found highly crucial for reducing the likelihood of the code 'regression' and other types of problems.

5.6 Discussion and conclusions

We have seen that Graniitti provides a new computational edge on the topic of central diffraction, accelerating the progress on the road towards understanding the topic from the first principles, whatever they are in the future. This concrete code, hopefully, also demystifies several aspects of the expert literature. We simulated how the glueball filter observable is being driven by the spin polarization components of the resonance spin density matrix, thus providing an ansatz that $J = 2$ glueballs produced in central production could be produced with different polarization compared with reasonably established tensor mesons such as $f_2(1270)$ produced most probably in pure transverse $|J_z| = 2$ polarization, which is our hypothesis – of course a mixture of pure states is possible, also with off-diagonal density matrix elements.

For the higher multiplicity final states, we point out here that the 'parallel processes' of simultaneous multiple pomeron-pomeron fusion are interesting. In principle, one could assume these to be distributed according to a Poisson distribution in the first approximation. We see that the 'serial process' of peripheral ladder exchange, which we have implemented, generates a cross section orders of magnitude lower than what is measured for the $\pi^+\pi^-\pi^+\pi^-$ in the preliminary ATLAS measurement [94] or seen in ALICE data [148]. Within the ALICE fiducial phase space at $\sqrt{s} = 7$ TeV, the cross section for the two body, four body and six body central states with double rapidity gap veto scales approximately $\propto 3^{-N}$, with $N = 2, 4, 6, \ldots$. The peripheral ladder exchange simulation has more like $\propto 10^{-N}$ scaling in this phase space, unless the ladder has some unknown non-perturbative mechanism enhancing the couplings or modifying the form factors within ladder vertices. There is always the possibility that the production of intermediate enigmatic $f_0(500)$ mesons and their sequential decays, which is readily available within the generator, would provide the explanation. Discriminating experimentally between the parallel, serial and sequential production is non-trivial, but possible based on the final state kinematics and higher dimensional statistical techniques. We emphasize the need for rigorous (multidimensional) fiducial measurements: cut and count and efficiency correct with possible generalized unfolding. Model-based interpretations or fits of data are always of limited use for the theory development, such will be any attempts to remove any physics background in the soft domain. Such subtractions can be done as a bonus exercise.

The tensor pomeron model predicts a distinctive 'dip' in the rapidity separation of a central proton-antiproton production, due to the fermion spin-1/2 structure. In essence, $\hat{t}$- and $\hat{u}$-sub-amplitudes have a negative relative sign in this case. Unfortunately, we see that this dip can be destroyed to become unobservable by the screening

loop but also by certain modifications of the unknown internal form factors. The destructive effect of the screening loop can be readily simulated with Graniitti. In general, it will be interesting to understand what is the detailed mechanism behind the baryon pair production in central production, what are the effective degrees of freedom and whether models such as Lund strings or other QCD motivated pictures can provide better descriptions.

Regarding spin, Graniitti is currently the only event generator that can generate arbitrary spin-dependent scattering amplitudes in low-mass central production. For the spin analyses, our engine puts the fiducial acceptance inversion on a rigorous footing. We demonstrated the 'implicit complexity' of the Lorentz rest frames measured by non-zero coefficients of the acceptance decomposition. This is shown in Figures B.1, B.2 and B.3. Unsurprisingly, the most simple one by this first-order measure is the direct Center of Momentum (CM) rest frame without any rotation from the lab and the most complex one seems to be the Pseudo-Gottfried-Jackson frame (PG). We recommend to implement and publish the experimental analysis in several different frames, which makes the measurements more future proof. The full dataset together with detector simulations should be made eventually available at portals such as opendata.cern.ch, because this allows re-studying the higher dimensional kinematic correlations and arbitrary observables. After all, even the spherical harmonic decomposition is only a projection to a certain basis and frame. We probed also dualities between central and forward observables. This is especially interesting for the ALICE and LHCb cases which do not measure forward protons.

We shall here shortly mention the related open inverse problems. What is the optimal tensor basis, not necessarily the spherical tensor basis, which would allow reconstructing the density matrix ρ given the measured set of moments t_{lm}? The formulation of this problem was proposed originally by Pauli for spin-1/2, which has a particularly simple solution in terms of Pauli matrices. A related question is what is the 'best way' to solve amplitudes in the partial wave basis from the measured spherical moments, a problem which is known to have polynomial ambiguities in numerous solutions [135]. Finding out a single physical solution is ill-posed for the case of overlapping resonances and continuum processes, which sum coherently at the amplitude level. The partial wave problem may be, however, actually more tractable for the full $2 \to 4$ process. To characterize the intrinsic invertibility of the detector acceptance through spherical harmonic expansion, we proposed calculating the condition number of the detector acceptance moment mixing matrix through singular value decomposition. The measured and inverted moments in multiple Lorentz frames, as we showed, give directly useful information for the theory development, independent of the underlying models. Also, Graniitti is fully equipped to fit the

parameters of the Tensor pomeron model against spherical moment expansions or any other observables – this requires only some CPU time. Finally, it would be interesting to construct maximally model independent and fully covariant spin measures from the frame dependent moment set.

In the future, one could include more scattering amplitudes, lattice simulations fused together with generative machine learning techniques for event-by-event proton structure fluctuations which would be a truly novel case, higher dimensional spin analysis algorithms and deep learning driven ultra efficient Monte Carlo importance sampling.

6 Combinatorial Superstatistics for Soft QCD

High energy diffraction and soft QCD span exciting final state topologies and fluctuations which have not yet been measured or characterized in a fully exhaustive way. In this work, we go beyond the standard measures and formulate a new framework to generalize rapidity gap counting with an emphasis on abstract structure, fiducial observables, experimental limitations and factorizing model dependence. The construction is based on a higher dimensional Bernoulli vector statistics with a combinatorial incidence algebra structure, independent of the underlying field theory. This is the first experimentally feasible framework designed to probe the seemingly arcane sign alternating scattering amplitude cutting rules of AGK-style, which are of interest in Regge theory, perturbative QCD and even in stringy black hole calculus. As an additional novel show case, we pose, construct and solve a highly related combinatorial stochastic superposition Poisson inverse problem using the Möbius inversion theorem.

Chapter in: arXiv:1910.06279 [hep-ph]

6.1 Introduction

The physics signatures used in searching and analysing interactions at high energy colliders are often based on specific final state 'topologies'. Examples of these span from the soft diffractive QCD processes, where different event classes are identified by detector responses in pseudorapidity space to new physics searches spanning complex event topologies with jets, leptons and missing transverse energy. In this work, we represent the final state topologies using binary vector spaces. This allows us to use powerful mathematical tools of combinatorics, namely, the incidence algebras introduced by G.C. Rota [149].

The combinatorial problems in high energy physics are numerous. To name a few, they span from combinatorial symmetry factors and Feynman diagram enumeration in QFT, Wick's theorem (Isserlis' theorem), Gribov-Glauber cancellations in heavy ion physics, Abramovski-Gribov-Kancheli cutting rules in Regge and field theory, numerous experimental combinatorial puzzles in track fitting and primary vertex association and in the resonance search analysis of final states with combinatorial background. Here we take a slightly different angle on this topic than what has been done previously. From the experimental point of view, the framework here can be utilized for the extraction of diffractive cross sections and completely novel fiducial measurements, and can be considered as a complete generalization of the double diffractive measurement done by the TOTEM experiment [38]. The approach developed in this work is the first explicit utilization of the Möbius inversion theorem in high energy physics, as far as we know. Perhaps the most well known Möbius inversion paper in physics is by Chen [150]. Kowalski, Polyzou and Redish used these tools in the context of partition combinatorics of a non-relativistic multiparticle scattering operator formalism [151]. The connection to supersymmetric mathematical physics models was first developed in papers by Spector [152] and Julia [153], and later by Onofri, Veneziano and Wosiek, which they called 'supercombinatorics' [154].

In soft QCD, different minimum bias processes have very different final state configurations over rapidity. However, when the luminosity grows high, nearly all capabilities to study diffractive processes are lost due to the multiple proton-proton interactions per bunch crossing – the pileup. This can be both on-time and off-time pileup, that is, it can propagate from the same or previous bunch cross due to the short 25 ns spacing at the LHC and long detector integration time windows. As an illustrative example, when the average number of inelastic interactions is $\mu \sim 10$, the so-called large rapidity gap events vanish almost completely when we talk about low-p_T events, and we are left only with uniform event signatures as our observables over pseudorapidity. Thus, there are naturally some fundamental limitations in how

large the luminosity can be even without considering all instrumentation and detector level complications. The optimal running setup depends heavily on the physics signal and background. The pile-up rates, being an order of 30-50 in ATLAS and CMS, even higher in the future, pose a major challenge for physics analysis, cross section measurements and searches for new physics phenomena. However, some high-p_T channels are still experimentally manageable up to $\mu \sim \mathcal{O}(10^2)$ or more. Thus we point out that one practical application of our work is the inversion of (minimum bias) trigger combinations under pileup, which was treated in work by ATLAS [155] and ALICE [156] collaborations with less powerful tools. In our other paper [157], we solve a related inverse problem of multiplicity distributions undergoing a compound Poisson autoconvolution process, by fusing techniques of characteristic functions and algorithms.

Section 6.2 starts with multivariate Bernoulli observables together with vector spaces over the finite Galois field GF(2) and sets them in the context of high energy diffraction. In Section 6.3 we go through the partially ordered sets, incidence algebras and the Möbius inversion and Section 6.4 is devoted for concrete measurements of observables. The other half of the paper deals with the combinatorial superposition problem in a compound Poisson scenario. Section 6.5 constructs the compound Poisson problem and a mathematical model is built, which is then solved using the combinatorial techniques from the first half of the paper. Finally in Section 6.6, we study the inverse problem numerically and summarize in Section 6.7. The appendix provides supplementary information.

6.2 Binary vector spaces and Diffraction

For the algebraic construction of Bernoulli observables, we need a vector space $\mathbb{F}_2^N$ over the finite Galois field GF(2) with addition as a component wise Boolean OR ($\vee$) and multiplication as a component wise Boolean AND ($\wedge$). As the inner product we can use $\langle \mathbf{a}|\mathbf{b}\rangle = \bigvee_{i=1}^N a_i \wedge b_i \in \mathbb{F}_2$, where $\mathbf{a}, \mathbf{b} \in \mathbb{F}_2^N$. The vector space is spanned by N standard basis vectors as $\text{span}(\mathbb{F}_2^N) = \{\mathbf{e}_k\}_{k=1}^N$ and the number of elements in an N-dimensional binary vector space is $|\mathbb{F}_2^N| = 2^N$, illustrated in Figure 6.2. The total number of subspaces is given by a sum over q-binomial coefficients, given in Appendix C.1.

The most useful property concerning the physics involved here is that the subspaces form an orthomodular poset (partially ordered set). Another aspect is that Boolean operators which are diagonal in one basis are diagonal in all bases. Projection operators are always diagonal, which is semi-intuitive if one thinks about the N-dimensional unit-hypercube structure. In [158] it was pointed out that the multivariate Bernoulli can be considered as a certain generalization of the Ising model. However, here we apply these to high energy physics.

Bernoulli random variables

Every component observable of vectors in $\mathbb{F}_2^N$ is treated as a Bernoulli random variable B_i with $1 \leq i \leq N$. The random variables B_i follow the distribution

$$\text{Ber}(B_i|m_i) = P_B(B_i = b|m_i) = m_i^b(1-m_i)^{1-b}, \tag{6.1}$$

where $b \in \{0, 1\}$. The mean is m_i and variance $\text{Var}[B_i] = m_i(1-m_i)$. Together, these can be encapsulated with a multivariate Bernoulli distribution $P(B_1 = b_1, B_2 = b_2, \ldots, B_N = b_N|\theta)$. The full distribution is either described directly with $2^N - 1$ parameters (-1 from normalization) associated with every 2^N elements of the binary vector space, in what we may call the *fundamental* or *natural* representation, or with parameters describing the expectation values and multipoint correlations between different Bernoulli variables, what we may call the *correlation* representations. We use the notation and representations constructed by Teugels [159].

Example $N = 3$: A centralized correlation representation is

$$m_i = \langle B_i \rangle, \quad i = 1, 2, 3 \tag{6.2}$$

$$\theta_{ij} = \text{Cov}[B_i, B_j] = \langle (B_i - m_i)(B_j - m_j) \rangle \tag{6.3}$$

$$= \langle B_i B_j \rangle - \langle B_i \rangle \langle B_j \rangle, \quad (i,j) \in \{(1,2), (1,3), (2,3)\} \tag{6.4}$$

$$\theta_{123} = \langle (B_1 - m_1)(B_2 - m_2)(B_3 - m_3) \rangle, \tag{6.5}$$

where the moments or correlation functions are centralized, that is, the mean is subtracted. The structure is only slightly more complicated for larger values of N, and is described in the following chapters. One interesting property is that for the multivariate Bernoulli distribution component variables which are pairwise uncorrelated $\mathrm{Cov}[B_i, B_j] = 0$ are also always independent, that is, the density factorizes $P(B_i, B_j) = P(B_i)P(B_j)$ [158]. This does not hold in general for multivariate probability distributions.

Probabilities and cross sections

In the fundamental representation, a probability vector $\mathbf{p}$ encapsulates the probabilities of *mutually exclusive* combinations of observables in the binary vector space

$$\mathbf{p} = (P_{\mathbf{b}_0}, P_{\mathbf{b}_1}, \ldots, P_{\mathbf{b}_{2^N-1}})^T, \quad \sum_c p_c = 1 \tag{6.6}$$

where indices go through all the elements of the binary vector space. The one-to-one correspondence is given uniquely by the binary expansion[1]

$$c = \sum_{i=1}^{N} b_i 2^{i-1} \;\Leftrightarrow\; \mathbf{b}^c = (b_1, b_2, \ldots, b_N), \tag{6.7}$$

between the index $0 \leq c \leq 2^N - 1$ and the binary vector $\mathbf{b}^c \in \mathbb{F}_2^N$.

Now turning into physical observables, the differential cross section element is given by the usual

$$d\sigma_{n'} = \frac{1}{F}\frac{1}{S_{n'}}\, d\Pi_{n'} |\mathcal{M}_{i\to n'}|^2, \tag{6.8}$$

where the matrix element squared $|\mathcal{M}_{i\to n'}|^2$ is the probability amplitude for the transition from the initial state to the n'-particle final state, encapsulating all dynamics. The $S_{n'}$ is a QFT combinatorial symmetry factor for identical final states, $F = 4\sqrt{(p_1 \cdot p_2)^2 - m_1^2 m_2^2} \longrightarrow 2s$ (high energy limit) is the incoming invariant Möller flux and the standard Lorentz invariant phase space element is

$$d\Pi_{n'} = (2\pi)^4 \prod_{j=1}^{n'} \frac{d^3 k_j}{(2\pi)^3 2k_j^0}. \tag{6.9}$$

[1] Here we remark on the difference between left and right ordered bit strings. We use the convention of most significant 'bit' (MSB) on *right*. In the most significant 'bit' on *left* representation, this expansion would be $c = \sum_{i=1}^{N} b_i 2^{d-i}$.

We now divide the fiducial phase space seen by the detector into N elements. Using the combinatorial incidence algebra notation, then the so-called 2^N partial cross sections can be expressed with

$$\begin{aligned}\boldsymbol{\sigma} = \frac{1}{F}\sum_{n'}\frac{1}{S_{n'}}\int d\Pi_{n'}\delta^{(4)}\left(p_1+p_2-\sum_{n'}k_j\right)|\mathcal{M}_{2\to n'}|^2\begin{pmatrix}1 & -1\\ 0 & 1\end{pmatrix}^{\otimes N}\\ \begin{pmatrix}1\\ A\{\{k_j\};\Xi_N\}\end{pmatrix}\otimes\\ \begin{pmatrix}1\\ A\{\{k_j\};\Xi_{N-1}\}\end{pmatrix}\otimes\cdots\otimes\begin{pmatrix}1\\ A\{\{k_j\};\Xi_1\}\end{pmatrix}.\end{aligned} \tag{6.10}$$

The expression above is a 2^N-vector, where the component σ_0 denotes the null part outside the total fiducial phase space. The acceptance (indicator) functions $A:\{k_j\}\to\{0,1\}$ encoding the Bernoulli trials are

$$A_i(\{k_j\};\Xi_i) = \begin{cases}1, & \text{if } \exists k_j\in\Xi_i\\ 0, & \text{otherwise.}\end{cases} \tag{6.11}$$

The acceptance function returns 1 if any of the final state particles was inside the phase-space domain Ξ_i defined for B_i and returns 0, if none were inside the domain. This function can be constructed with simple phase space cuts suitable for analytical calculations. In practice, they are most easily calculated with soft QCD Monte Carlo event generators, where arbitrary fiducial phase space cuts are possible. We abstract out the detailed singularity and theory structure connections with the phase space cuts out at this point, they are theory specific, but certainly collinear and infrared singularities may be used to design a specific structure for the acceptance functions.

Experimental efficiency and resolution functions work as a mixing map $\sigma_c\mapsto\sigma_{c'}$, which is a highly complicated stochastic non-linear operator modulating the pure acceptance indicator function. For the expectation values, it can be represented with a finite linear *folding* operator acting on the final states partial cross section vector as $\mathcal{F}:\boldsymbol{\sigma}\to\tilde{\boldsymbol{\sigma}}$, where $\boldsymbol{\sigma}$ denotes the partial cross sections at the fiducial phase space particle level and $\tilde{\boldsymbol{\sigma}}$ denotes the visible partial cross sections at the detector level. Both can have almost arbitrary definitions, depending on the problem. The hierarchy goes as follow

$$d\mathcal{O}\xrightarrow[\forall k]{\sum_{n'}\int_{\Omega_{n'}}}\boldsymbol{\sigma}\xrightarrow[-\sigma_E,+\sigma_L]{\mathcal{F}}\tilde{\boldsymbol{\sigma}}, \tag{6.12}$$

where the first map is the 'theory' or 'generator' and the second one is the 'detector', to illustrate. The zero-th component σ_0 of $\boldsymbol{\sigma}$ denotes the invisible fiducial cross section in the fiducial phase space division (geometrically invisible), only accessible theoretically or via indirect measurements. The inverse operator $\mathcal{F}^{-1}$ is the so-called *unfolding* process, which can be naively solved with a $2^N - 1$ matrix inversion, obtained via detector simulation. The σ_E denotes the integrated efficiency loss, which is not the same as pure acceptance loss σ_0. Integrated leakage from outside the fiducial to inside the visible is denoted with σ_L, which may happen due to e.g. material re-scattering – this can be a non-negligible effect with very forward detectors. We emphasize here that the unfolding is for efficiency and smearing induced corrections, not to extrapolate the detector geometry.

As an example of the indicator functions one could consider a simple division of the (pseudo)rapidity into N intervals with suitable p_t thresholds per interval. Experimentally, the intervals may overlap over pseudorapidity such that $[\eta_{i,\min}, \eta_{i,\max}] \cup [\eta_{j,\min}, \eta_{j,\max}] \neq \{0\}$ for a detector pair (ij), and thus making them trivially correlated. It is possible, in principle, via suitable non-overlapping fiducial definition to take this into account in the unfolding procedure which de-correlates the data. Another option is simply to take care that no overlap happens physically. Also, the detectors may have gaps over η such that $[\eta_{i,\min}, \eta_{i,\max}] \cup [\eta_{j,\min}, \eta_{j,\max}] = \{0\}$. However, it is worth pointing out that the formalism presented here does not assume statistical independence of any kind, either implicit or explicit.

Abramovski-Gribov-Kancheli cutting rules

We collect here some illustrating results of the AGK cutting rule calculus [25]. By no means this is a complete description, but rather of an introductory nature. These rules are somewhat similar to more well known Cutkosky-Landau cutting rules based on the S-matrix unitarity, the rules which treat scattering amplitude discontinuities. The AGK rules were introduced in the Regge theory context, but they are actually flexible in terms of the underlying theory. What we need here is the reggeization, which happens in field theories under Regge limit but also in string theory. The AGK calculus results here are collected from a stringy black hole context [160] and from QCD papers [161, 162]. We assume the usual unitary $\mathcal{S}$-matrix and $\mathcal{T}$ matrix related with

$$\mathcal{S} = 1 + i\mathcal{T}. \tag{6.13}$$

Let us have the elastic scattering amplitude $\mathcal{T}_{AB}(s,t)$ with a Sommerfield-Watson integral transform representation in the complex angular momentum space as

$$\mathcal{T}_{AB}(s,t) = \int \frac{d\omega}{2i}\xi(\omega)s^{1+\omega}\mathcal{F}(\omega,t) \;\text{ with }\; \xi(\omega) = \frac{\tau - e^{-i\pi\omega}}{\sin\pi\omega}, \tag{6.14}$$

where $\mathcal{F}(\omega,t)$ is the corresponding partial wave, $\omega = J-1 \in \mathbb{C}$ being the conjugate variable to s together with the signature factor ξ for the positive or negative signature $\tau = \pm 1$. The positive signature case, such as Pomeron exchange, can be written more compactly as $\xi(\omega) = i + \tan(\frac{\pi}{2}\omega)$. Now assume an exchange of n Reggeons, that is, exchange of Regge trajectories.

The n-cut discontinuity over ω of $\mathcal{F}(\omega,t)$ can be written in terms of vertex functions $\mathcal{V}_n^a, \mathcal{V}_n^b$ and integrating over transverse momentum phase space as [162]

$$\text{DISC}_\omega^{(n)}[\mathcal{F}(\omega,t)] = 2\pi i \int \frac{d\Omega_n}{n!}\delta(\omega - \sum_j \beta_j)\Pi(\{\beta_j\})\mathcal{V}_n^a(\{\mathbf{k}_j,\omega\})\mathcal{V}_n^b(\{\mathbf{k}_j,\omega\}) \tag{6.15}$$

$$d\Omega_n \equiv (2\pi)^2\delta^{(2)}\left(\mathbf{q} - \sum_{j=1}^n \mathbf{k}_j\right)\prod_{j=1}^n \frac{d^2\mathbf{k}_j}{(2\pi)^2} \tag{6.16}$$

$$\Pi(\{\beta_j\}) \equiv \text{Im}\left[-i\prod_j i\xi_j\right] = (-1)^{n-1}\frac{\cos\left[\frac{\pi}{2}\sum_j(\beta_j + \frac{1+\tau_j}{2}\right]}{\prod_j \cos\left[\frac{\pi}{2}(\beta_j + \frac{1-\tau_j}{2})\right]}, \tag{6.17}$$

where $\beta(-\mathbf{k}^2) \equiv \alpha(-\mathbf{k}^2) - 1$ with $\alpha(t_j = -\mathbf{k}_j^2)$ being the Regge trajectory function, for the soft Pomeron often used parametrization is $\alpha(t) \simeq 1.08 + 0.25t$. The factor $n!$ comes from all possible exchange diagram planar and crossed non-planar orderings.

Now the amplitude for the n-cut exhange can be written using the Sommerfield-Watson transform and the cut discontinuity as [161]

$$\mathcal{T}_{AB}^{n-\text{cut}}(s,t) = \int \frac{d\omega}{2i}\xi(\omega)s^{1+\omega}\text{DISC}_\omega^{(n)}[\mathcal{F}(\omega,t)] \tag{6.18}$$

$$= \pi\int \frac{d\Omega_n}{n!}\xi(\sum_j \beta_j)s^{1+\sum_j \beta_j}\Pi(\{\beta_j\})\mathcal{V}_n^a(\{\mathbf{k}_j,\sum_i \beta_j\})\mathcal{V}_n^b(\{\mathbf{k}_j,\sum_j \beta_j\}). \tag{6.19}$$

Next we write down the Abramovski-Gribov-Kancheli sign alternating combinatorial factors [162]

$$\boxed{G_k^n = \begin{cases} 2^{n-1}(-1)^n + \Pi(\{\beta_j\}), \text{ for } k=0 \text{ cut Pomerons}, \\ 2^{n-1}(-1)^{n-k}\frac{n!}{(n-k)!k!}, \text{ for } 0 < k \leq n, \end{cases}} \tag{6.20}$$

n,k	0	1	2	3	4	5	6
1	0	1	0	0	0	0	0
2	1	-4	2	0	0	0	0
3	-3	12	-12	4	0	0	0
4	7	-32	48	-32	8	0	0
5	-15	80	-160	160	-80	16	0
6	31	-192	480	-640	480	-192	32

Table 6.1: AGK factors G_k^n for the even signature (Pomeron) case with n exchanges and k of them cut.

which we tabulate for illustration in Table 6.1.

Let us now point some integer sequence properties of the AGK series, which might be illuminating in our hyperspace context. The series $|G_{k=1}^n|$ is the number of edges in an n-hypercube. The series $|G_{k=2}^n|$ is the number of diagonals in an n-hypercube of length $\sqrt{2}$. The series $|G_{k=3}^n|$ is the number of 3-cycles in the halved n-cube graph. These suggest to us directly that in order to be (experimentally) sensitive to n Pomeron exchanges, the number of rapidity slices N should be at least $n+1$. Now the use of AGK cutting factors is that they allow us to represent the total discontinuity over s-space scattering amplitude as a weighted sum over all k-cut Pomeron contributions [162]

$$\mathcal{A}^n(s,t) \equiv \mathrm{DISC}_s[\mathcal{T}_{AB}^{n-\mathrm{cut}}(s,t)] = \sum_{k=0}^{n} a_k^n(s,t), \quad \text{where}$$
$$a_k^n = 2\pi i\, G_k^n \int \frac{d\Omega_n}{n!} s^{1+\sum_i \beta_i} \mathcal{V}_n^a(\{\mathbf{k}_j, \sum_i \beta_i\}) \mathcal{V}_n^b(\{\mathbf{k}_j, \sum_i \beta_i\}). \tag{6.21}$$

The last step in our short discussion on this topic is the impact parameter b space picture and eikonalization. The eikonal opacity function in the impact parameter space is obtained using the (inverse) Fourier transform of a single discontinuity integrated over ω [162]

$$\Omega(s,b) \equiv \frac{1}{\pi i} \int \frac{d^2\mathbf{q}}{(2\pi)^2} e^{-i\mathbf{b}\cdot\mathbf{q}} \int d\omega\, \mathrm{DISC}_\omega[\mathcal{F}(\omega, -\mathbf{q}^2)]\, s^\omega \tag{6.22}$$

with the amplitude recovered in the s-space by (forward) Fourier transform of the n-times exponentiated amplitude, by *assuming factorization* of each exchange as

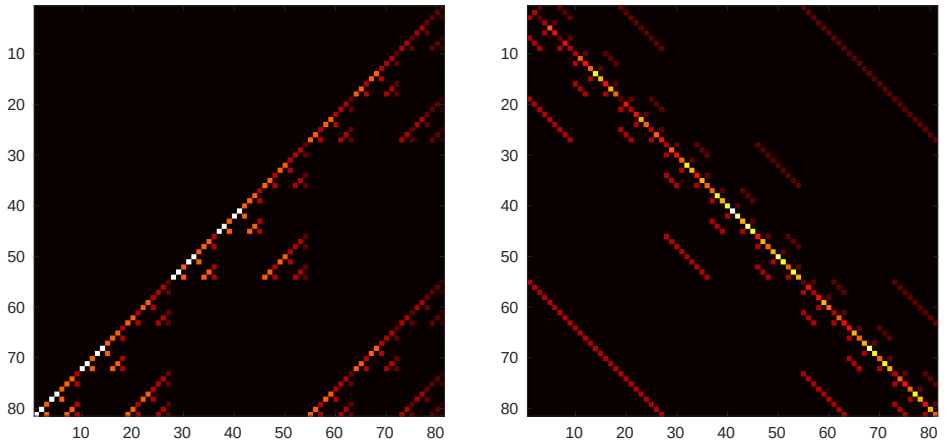

Figure 6.1: A tensor product chain $M^{\otimes 4}$ on the left and a tensor sum chain $\frac{1}{4}M^{\oplus 4}$ on the right. Black to red to white denote values in [0,1].

$\mathcal{V}_n^a = [\mathcal{V}_1]^n$, giving

$$\mathcal{A}_k^n(s,t) = 4is\frac{(-1)^{n-k}}{k!(n-k)!}\int d^2\mathbf{b}e^{i\mathbf{b}\cdot\mathbf{q}}\left[\Omega(s,b)\right]^n \tag{6.23}$$

with summation $n \geq k$ resulting in

$$\mathcal{A}_k(s,t) = 4is\int d^2\mathbf{b}e^{i\mathbf{b}\cdot\mathbf{q}}P(k,s,b). \tag{6.24}$$

Where the probability is a sum over the $\mathcal{S}$-matrix elements [162]

$$P(k,s,b) \equiv \sum_{n\geq k}[\mathcal{S}(s,b)\mathcal{S}(s,b)^\dagger]_k^n = \frac{[\Omega(s,b)]^k}{k!}e^{-\Omega(s,b)}, \tag{6.25}$$

which gives a Poisson distribution with $\langle k(s,b)\rangle = \mathrm{Var}[k(s,b)] = \Omega(s,b)$. That is, the number of k-cut Pomerons at given energy squared s and impact parameter b is Poisson distributed in this picture. Now, in typical Monte Carlo implementations, one needs also the parton densities to pick the longitudinal momentum fractions from vertex factors $\mathcal{V}_1^{a,b}$ for each exchanged Pomeron and at least triple-Pomeron interactions with diffractive cuts, in addition (hard) QCD matrix elements, parton shower and fragmentation.

As an example, we shall refer to the description of QGSJet-II event generator for details [163], which implements advanced Reggeon calculus for soft and hard interactions. The scope of this paper is to provide new algebraic methods for measuring the observables. We use the Poissonian AGK distribution as our motivation for the later construction. We shall also note here that a left stochastic matrix devised by Ryskin and Bartels [161], describes the 'non-diagonal' cut transitions necessary for rapidity gap configurations

$$M = \begin{pmatrix} 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 \\ 1 & 0 & \frac{1}{2} \end{pmatrix}, \tag{6.26}$$

which has $\mathbf{x} = [1, -4, 2]^T$ as its eigenvector, the first non-trivial row from Table 6.1. That is, for every rapidity interval, these ratios hold, and the transition matrix describes the stochastic probability of transition from different configurations per chain. In Figure 6.1 we illustrate the nature of this matrix under the Kronecker product $A \otimes B$ and the Kronecker sum defined as $A \oplus B = A \otimes I_A + I_B \otimes B$, where I is an identity matrix. The product matrix belongs to the class of inclusion-exclusion matrices whereas the sum matrix resembles a hypercube adjacency (connectivity) matrix. Later we learn how these concepts appear in the incidence algebra.

Rapidity gaps as a spacing distribution

It is typical that 'non-diffractive events' have an exponentially decreasing maximum rapidity gap distribution $\sim \exp(-\ell_c \Delta y)$, where the correlation length $\ell_c \sim 1$ depends on the average multiplicity density. This can be derived assuming n independent final state emissions over rapidity. Diffractive events are those which have 'anomalously' large gaps. The rapidity gap itself is a kinematic consequence given the excited (dissociative) proton system mass is small enough and the initial state boost large enough, however, the M^2 distribution itself is driven dynamically. The standard triple Pomeron limit distributions obtained via generalized multibody unitarity together with Regge asymptotics give $d\sigma/dM^2 \sim 1/(M^2)^{\alpha_P(0)}$, where the $\alpha_P(0)$ is the effective Pomeron intercept $\alpha_P(0) \equiv 1 + \Delta_P \sim 1.08$. Then by standard kinematic[2] change of a variable $\phi^{-1}(\Delta y) \equiv s \exp(-\Delta y) = M^2$, we obtain the single diffractive

[2] Beam rapidity is $y_\pm = \pm \ln(\sqrt{s}/m_p)$ and the diffractive system spans $y_{\text{span}} = \ln(M^2/m_p^2)$.

rapidity gap distribution

$$P_D(\Delta y) \sim \frac{d\sigma}{d\Delta y} = \frac{d\sigma}{dM^2}(\phi^{-1}(\Delta y)) \left|\frac{d\phi^{-1}}{d\Delta y}\right| = \frac{1}{(s\exp(-\Delta y))^{\alpha_P(0)}} \left|\frac{-s}{e^{\Delta y}}\right| = e^{\Delta y(\alpha_P(0)-1)}. \quad (6.27)$$

Above, we neglected Mandelstam t-dependence of the triple Regge expressions by taking $t \to 0$, but this is not relevant for this discussion. Now Equation 6.27 is a slowly rising exponential, instead of decreasing such as for the random non-diffractive processes. In the case of a double diffractive process, the equivalent derivation goes via the Jacobian determinant $\det J(\Delta y, y_0) = sm_p^2 e^{-\Delta y}$, from $\phi_i^{-1}(\Delta y, y_0) = (sm_p^2 e^{-\Delta y \pm 2y_0})^{1/2} = M_i^2$ with $i = 1, 2$ for the two systems and $\Delta_y^{DD} = -\ln\left(M_1^2 M_2^2/(sm_p^2)\right)$ and $y_0 = \frac{1}{2}\ln(M_1^2/M_2^2)$. This gives us

$$P_{DD}(\Delta y) \sim \frac{d\sigma}{d\Delta y dy_0} = \frac{d^2\sigma}{dM_1^2 dM_2^2}(\phi_{1,2}^{-1}(\Delta y, y_0))\,|\det(J)| = \prod_i \left[\frac{1}{(sm_p^2\exp(-\Delta y \pm 2y_0))^{\alpha_P(0)/2}}\right] \frac{sm_p^2}{e^{\Delta y}} \quad (6.28)$$

$$\sim e^{\Delta y(\alpha_P(0)-1)}, \quad (6.29)$$

where the dependence on y_0 vanishes and the functional dependence is the same as in the single diffractive case. The boundary conditions are different, because the both edges of the gap are moving in the case of double diffractive scattering. In general, the gap may be taken with respect to the detector geometric boundary or with respect to the minimum or maximum of the rapidity interval (type I), or we can count distances between particles (type II). One must remember that any unitarization scheme may modify these results. That is, the triple Pomeron expressions do not satisfy unitarity under $s \to \infty$, which is a long standing problem, with numerous partially plausible solutions (eikonalization, triple Pomeron decoupling, ...). Currently, there is no detailed data enough to gain new insights on this problem neither the non-perturbative QCD techniques are advanced enough to solve it. However, new LHC data may change the situation.

At this point, we refer the reader to mcplots.cern.ch for the measured rapidity gap distributions at the LHC with different minimum p_t thresholds and Monte Carlo event generator comparisons. The large (diffractive) gaps spanning several units of rapidity originate from charge-neutral exchanges, such as exchange of color neutral gluon systems (pomerons) or photons. The interaction rate of gap events may be

screened by additional elastic or inelastic interactions, this is known as the gap survival discussion. An important property is that measuring the final states down to an arbitrary infrared cutoff, not experimentally accessible, modifies the observed rates of rapidity gap events. This corresponds e.g. to extra emission of soft gluons. Thus, the rapidity gap events are not clearly 'infrared safe' without an explicit p_t threshold definitions. Later in this work, in Figure 6.6, we show with a toy Monte Carlo simulation how the combinatorial gap rates run as a function of p_t thresholds for four different particle densities per rapidity slice. We see that it is possible, given different p_t thresholds and particle densities, to run the relative event rates of different final state topologies to wildly different values.

For the non-diffractive events, now let $Y_0, Y_1, \ldots, Y_n$ be $n+1$ rapidity intervals spanned by an n particle final state proceed e.g. via 'string breaking process'. To calculate statistics of these, we may use the Darling's contour integral technique from mathematical statistics [164] via the Laplace transform

$$\mathbb{E}[f_0(Y_0)f_1(Y_1)\ldots f_n(Y_n)] = \frac{n!}{2\pi i}\int_{c-i\infty}^{c+i\infty} e^z \prod_{j=0}^{n}\int_0^\infty dz\, dr_j\, e^{-r_j z} f_j(r_j). \tag{6.30}$$

The path of the integral is along the straight line $\mathrm{Re}\{z\} = c$ and $f_j(x)$ are arbitrary real valued functions. Using this expectation integral, one can derive numerous results regarding the spacing statistics. For example:

MaxGap: The probability that all rapidity gaps are smaller than β is

$$P(Y_j < \beta\ \forall j) = \sum_{0\le j<1/\beta} \binom{n+1}{j} (-1)^j (1-\beta j)^n. \tag{6.31}$$

MinGap: The probability that all rapidity gaps are larger than α is

$$P(Y_j > \alpha\ \forall j) = (1-(n+1)\alpha)^n \quad \text{s.t.} \quad \alpha < 1/(n+1). \tag{6.32}$$

Both of these are derived in Darling's paper [164] under the uniform distribution assumptions. We see that given uniformly distributed rapidities, the number of final states gives physical boundaries for the probabilities, visible in the inequalities in Equation 6.31 and 6.32. Now clearly, one can consider higher-order spacing statistics such as distributions of the largest and the second largest gap and so on. Clearly, our incidence algebra construction is an integrating embedding of these taking also into account the translation (boost) position on the rapidity axis.

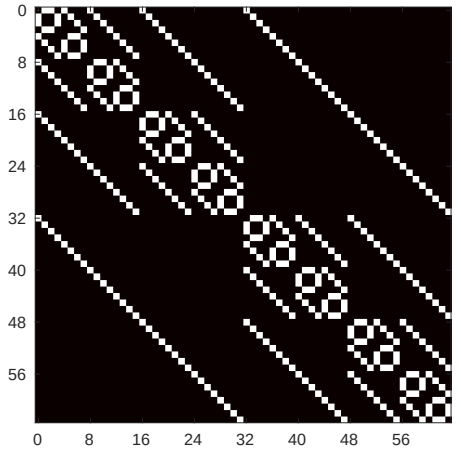

Figure 6.2: The binary vector space illustrated as an $N = 6$ hypercube graph with its corresponding adjacency matrix (black is 0, white is 1).

Our construction includes, in the limit $N \to \infty$, exactly the 'traditional' rapidity gap description. That is, one can recover the one dimensional rapidity gap distribution from the N-dimensional combinatorial distribution. Moreover, in practice, N needs not to be infinite due to the random correlation length over rapidity which smears out useful information at very small gap values.

Hypergraph and topology

In Figure 6.2 we demonstrate the vector space as an equivalent hypercube $G(E, V)$. Each vertex in set E corresponds to one particular final state configuration and partial cross section component. The number of edges in the graph is $N2^{N-1}$, also counting the number of non-zero phase space elements (bits). The total number of subspaces is 2825, for $N = 6$. That is, it is a radically richer space than a single dimensional projection of rapidity gap distributions. In the following sections, a complete algebraic construction for any finite N will be given.

We point out that our definition of 'event topology' is reasonably well defined even if slightly abstract at first sight. Regarding the topological invariants of the hypercube, the Euler characteristic gives us $V - E + F = 2 - 2\gamma(G_N)$ where F is the number of faces and the minimum genus of N-cube for $N \geq 2$ is [165]

$$\gamma(G_N) = (N - 4)2^{N-3} + 1. \tag{6.33}$$

That is, if $N \leq 3$, then $\gamma(G_N) = 0$ and we can embed the planar graph without edge crossings on a sphere, $N = 4$ requires a one hole torus and then exponentially more complicated topologies are generated. The connection between final state geometry (kinematics) and topology is now well defined – more complicated 'event topologies' require larger N. An interesting construction would be the limit $N \to \infty$, that is, from finite dimensional hypercubes to an infinite dimensional construction. In practice, approximation schemes such as neural networks could be useful proxies for that.

6.3 Posets, incidence algebra and Möbius inversion

To have the necessary mathematical preliminaries to attack our problems further, we need to first define here a few basic concepts well known in combinatorics. For more information, we refer the reader to the original papers on incidence algebras by G.C. Rota [149] and a textbook on combinatorics by R. Stanley [166].

Let P be a set. A binary relation $\leq$ between elements $x, y, z \in P$ satisfying

$$\text{1. Reflexivity: } x \leq x \text{ for all } x \in P \tag{6.34}$$

$$\text{2. Antisymmetry: if } x \leq y \text{ and } y \leq x, \text{ then } x = y \tag{6.35}$$

$$\text{3. Transitivity: if } x \leq y \text{ and } y \leq z, \text{ then } x \leq z \tag{6.36}$$

is called a *partial order*. Thus, a set equipped with a partial order is called a *poset*.

Now $(P, \leq)$ is a poset. An *incidence algebra* is constructed by first defining an interval $[x, y] = \{z \in P \,|\, x \leq z \leq y\}$. Then the incidence algebra on P is

$$I(P) = \{f : P \times P \to \mathbb{R} \,|\, f(x, y) = 0 \text{ unless } x \leq y\}. \tag{6.37}$$

Thus, it gives us the precise definition of the inclusion operation. The incidence algebra has two operations of addition and multiplication defined as

$$\text{1. Addition : } (f + g)(x, y) = f(x, y) + g(x, y) \tag{6.38}$$

$$\text{2. Multiplication : } (f * g)(x, y) = \sum_{x \leq z \leq y} f(x, z) g(z, y) \tag{6.39}$$

so the addition is point-wise and the 'multiplication' is a convolution sum. Thus, division can be understood as a deconvolution operation. Common examples of locally finite posets P are: the positive integers ordered by divisibility, an inclusive order of the subspaces of a vector space over a finite field (projective spaces) or all the inclusively ordered subsets of a given set.

In order to proceed, we need a ζ-function[3] defined as

$$\zeta(x, y) = \begin{cases} 1, & \text{if } x \leq y. \\ 0, & \text{otherwise.} \end{cases} \tag{6.40}$$

and its inverse ζ^{-1} is the so-called Möbius function u (as in number theory) defined

[3] This has a relation with the Riemann ζ-function, which explains the name.

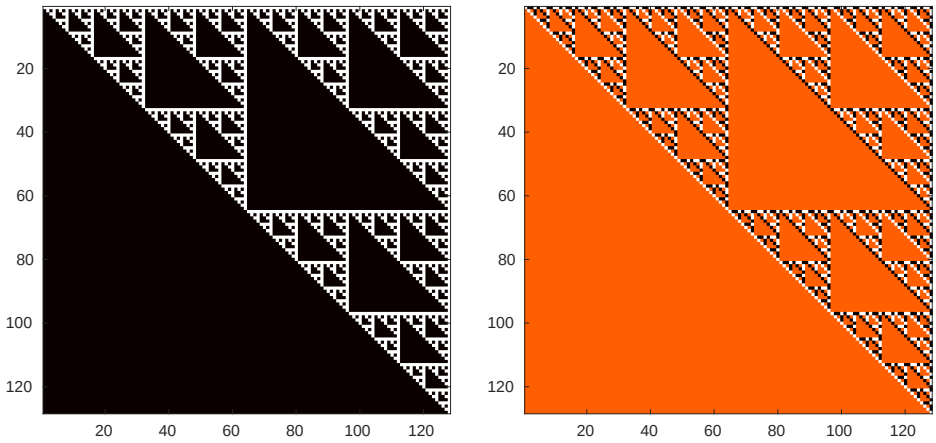

Figure 6.3: Matrices $\zeta_{128\times128}$ and $\zeta^{-1}_{128\times128}$ for $N = 7$. On the left black is 0 and white is 1. On the right, orange is 0, black is -1 and white is 1.

with a recursive relation

$$u(x,y) = \begin{cases} 1, & \text{if } x = y \\ -\sum_{x\leq z\leq y} u(x,z) & \text{for } x < y \\ 0, & \text{otherwise.} \end{cases} \tag{6.41}$$

The identity or delta-function works as expected in $I(P)$: $\delta(x,y) = 1$ if $x = y$, and 0 otherwise. In what follows, the ζ-function can be represented with an upper triangular matrix of size $2^N \times 2^N$ with Boolean elements, unit diagonal and determinant 1. Thus, the Möbius matrix is found easily by inverting the ζ-matrix which results in an upper triangular matrix with unit diagonal and alternating signed elements.

The Möbius inversion theorem

The *Möbius inversion theorem* goes as follows [166]. Let $(P, \leq)$ be a finite poset and f and g elements of the incidence algebra $I(P)$. The theorem states two equivalent

formulas in direct $g \mapsto f$ and inverse $f \mapsto g$ directions

$$f(x,y) = \sum_{x \le z \le y} g(x,z) \qquad \Leftrightarrow \qquad f = g * \zeta \tag{6.42}$$

$$g(x,y) = \sum_{x \le z \le y} f(x,z)u(z,y) \qquad \Leftrightarrow \qquad g = f * u. \tag{6.43}$$

Intuitively, multiplying by ζ can be understood as integration and by u as a differentiation operation. The same convolution products are easily written as

$$\mathbf{f} = \zeta \mathbf{g} \;\Leftrightarrow\; \mathbf{g} = \zeta^{-1}\mathbf{f}, \tag{6.44}$$

using a representation with column vectors $\mathbf{f}, \mathbf{g}$ and triangular square matrices ζ, ζ^{-1}. When increasing the dimension N, the recursive Sierpinski triangle fractal structure in these matrices will manifest itself. This is shown in Figure 6.3.

The principle of inclusion-exclusion

The Principle of Inclusion Exclusion (PIE) is the Möbius inversion for subsets [166]. The basic 'sieving' principle of PIE itself has been known for long. Now let different scattering cross section domains for each Bernoulli observable be represented with subsets $D_1, D_2, \ldots, D_N \subseteq D$. By defining two functions f and g on the poset $(P, \subseteq)$ having $|P| = 2^N$ and $\mathbb{F}_2^N \cong P$ as

$$f(\chi) = \left| \bigcap_{i \in \chi} D_i \right| \tag{6.45}$$

$$g(\chi) = \left| \{ x \in D \,|\, x \in D_i \text{ for all } i \in \chi \text{ holds } x \notin D_j \text{ for all } j \notin \chi \} \right|, \tag{6.46}$$

where $\chi \subseteq P$. This can be written in the forward direction with

$$f(\Upsilon) = \sum_{\chi \subseteq \Upsilon \subseteq P} g(\chi). \tag{6.47}$$

Now the inverse, the PIE, follows directly as

$$g(\chi) = \sum_{\chi \subseteq \Upsilon \subseteq P} (-1)^{|\Upsilon - \chi|} f(\Upsilon) \tag{6.48}$$

with the earlier introduced Möbius function

$$u(\chi, \Upsilon) = \begin{cases} (-1)^{|\Upsilon - \chi|}, & \text{if } \chi \subseteq \Upsilon \\ 0, & \text{otherwise.} \end{cases} \tag{6.49}$$

Next we apply these tools.

Algebraic representations

The vector valued Bernoulli distributions can be now written using

$$\mathbf{p} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}^{\otimes N} \mathbf{m} \Leftrightarrow \mathbf{m} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^{\otimes N} \mathbf{p}, \tag{6.50}$$

where we immediately recognize on left ζ^{-1} and on right the ζ-matrix obtained via recursive use of the Kronecker tensor product. This is the form given by Teugels in [159], but he did not point out the connection to the incidence algebras or Möbius functions. The form is very similar to ones often used in quantum mechanics with finite degrees of freedom, for example the Hilbert space decompositions into subsystems. Using the Kronecker tensor products, the probability vector $\mathbf{p}$ is

$$p_c = \left\langle \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}^{\otimes N} \begin{pmatrix} 1 \\ B_N \end{pmatrix} \otimes \begin{pmatrix} 1 \\ B_{N-1} \end{pmatrix} \otimes \cdots \otimes \begin{pmatrix} 1 \\ B_1 \end{pmatrix} \right\rangle_c . \tag{6.51}$$

The polynomial vector of *ordinary moments* $\mathbf{m}$ is

$$m_c = \left\langle \prod_{i=1}^{N} B_i^{b_i^c} \right\rangle = \left\langle \begin{pmatrix} 1 \\ B_N \end{pmatrix} \otimes \begin{pmatrix} 1 \\ B_{N-1} \end{pmatrix} \otimes \cdots \otimes \begin{pmatrix} 1 \\ B_1 \end{pmatrix} \right\rangle_c . \tag{6.52}$$

The polynomial vector of *central moments* $\boldsymbol{\delta}$ is

$$\begin{aligned} \delta_c &= \left\langle \prod_{i=1}^{N} (B_i - \langle B_i \rangle)^{b_i^c} \right\rangle \\ &= \left\langle \begin{pmatrix} 1 \\ B_N - \langle B_N \rangle \end{pmatrix} \otimes \begin{pmatrix} 1 \\ B_{N-1} - \langle B_{N-1} \rangle \end{pmatrix} \otimes \cdots \otimes \begin{pmatrix} 1 \\ B_1 - \langle B_1 \rangle \end{pmatrix} \right\rangle_c , \end{aligned} \tag{6.53}$$

where c is given by the binary expansion, $0 \leq c \leq 2^N - 1$ and $b_i^c \in \{0, 1\}$ of $\mathbf{b}^c$. The central moments describe the multipoint correlations ($\# 2^N - N - 1$) between any 2 or more subspaces in phase-space (such as rapidity intervals) and B_i are the corresponding Bernoulli random variables. The correlation functions $\langle B_i \dots B_j \rangle$ factorize to a product of $\langle B_i \rangle \dots \langle B_j \rangle$ if no correlations are present, as usual. This property can be used to test factorization of physics, such as factorization properties of (soft) QCD amplitudes.

Subspace decompositions

Now we drop the zero vector case $c = 0$, which corresponds to the case $B_1 = B_2 = \dots B_N = 0$. This gives no contribution being the additive identity of the algebra, physically outside the fiducial phase space of the definition. Thus the vector $\mathbf{p}$ will be then $2^N - 1$ dimensional, and $\sum p_c = 1$ still by definition. By a slight abuse of notation, we will continue with this convention for the rest of the paper unless otherwise mentioned. Now analogously to Equation 6.50, we write

$$\Lambda \mathbf{p} = \mathbf{r}, \tag{6.54}$$

where $\mathbf{r}$ is an auxiliary vector which collects the additive subspace probabilities of different mutually exclusive final states and Λ is a full rank (invertible) square Boolean matrix of size $2^N - 1$. The number of different subspaces of varying dimensionality is given by q-binomials of Equation C.1 in Appendix C.1.

Example $N = 2$: Denote the Bernoulli probability of the observable B_1 with P_{B_1} and of B_2 with P_{B_2}. The probability of events obeying fiducial binary combinations $\mathbf{b}_1 = (1,0) \equiv [B_1] \wedge [\neg B_2]$, $\mathbf{b}_2 = (0,1) \equiv [\neg B_1] \wedge [B_2]$ and $\mathbf{b}_3 = (1,1) \equiv [B_1] \wedge [B_2]$ can be obtained by inverting a system

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} P_{1,0} \\ P_{0,1} \\ P_{1,1} \end{pmatrix} = \begin{pmatrix} P_{B_1} \\ P_{B_2} \\ P_{B_1 \vee B_2} = P_{B_1} + P_{B_2} - P_{B_1 \wedge B_2} \equiv 1 \end{pmatrix}, \tag{6.55}$$

$$\Lambda_{3\times 3}^{-1} = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 0 & 1 \\ 1 & 1 & -1 \end{pmatrix}. \tag{6.56}$$

Example $N = 3$: Observables are now B_1, B_2, B_3. We get in a completely analogous way as in the $N = 2$ case

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} P_{1,0,0} \\ P_{0,1,0} \\ P_{1,1,0} \\ P_{0,0,1} \\ P_{1,0,1} \\ P_{0,1,1} \\ P_{1,1,1} \end{pmatrix} = \begin{pmatrix} P_{B_1} \\ P_{B_2} \\ P_{B_3} \\ P_{B_1 \vee B_2} = P_{B_1} + P_{B_2} - P_{B_1 \wedge B_2} \\ P_{B_1 \vee B_3} = P_{B_1} + P_{B_3} - P_{B_1 \wedge B_3} \\ P_{B_3 \vee B_2} = P_{B_2} + P_{B_3} - P_{B_2 \wedge B_3} \\ P_{B_1 \vee B_2 \vee B_3} = P_{B_1} + P_{B_2} + P_{B_3} - Z \equiv 1 \end{pmatrix}, \tag{6.57}$$

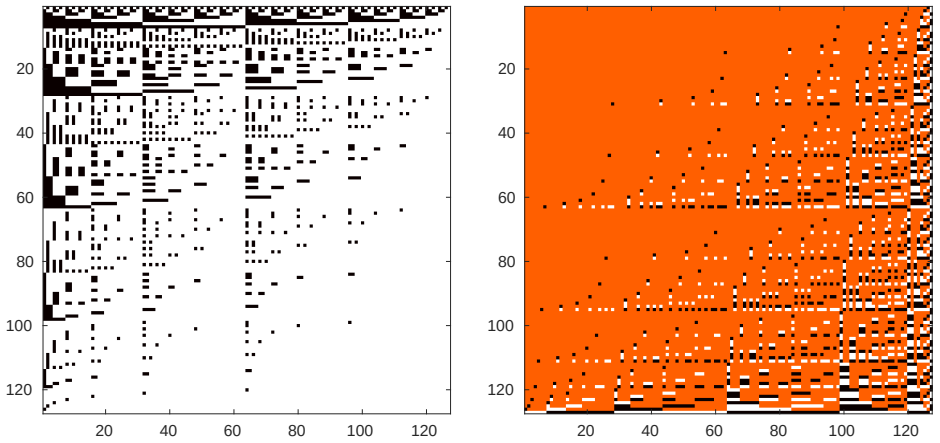

Figure 6.4: Matrices $\Lambda_{127\times127}$ and $\Lambda^{-1}_{127\times127}$ for $N = 7$. On the left black is 0 and white is 1. On the right, orange is 0, black is -1 and white is 1.

$$\Lambda^{-1}_{7\times7} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 0 & 1 & 1 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 1 & 0 & 1 & -1 \\ -1 & 0 & 0 & 1 & 1 & 0 & -1 \\ 1 & 1 & 1 & -1 & -1 & -1 & 1 \end{pmatrix}, \tag{6.58}$$

where $Z = P_{B_1\wedge B_2} + P_{B_1\wedge B_3} + P_{B_2\wedge B_3} - P_{B_1\wedge B_2\wedge B_3}$.

We can proceed in a same way up to an arbitrary number N of Bernoulli observables, which span a number of $2^N - 1$ non-zero combinations. To summarize: the idea is to generate a system of all linear combinations of event rates described by a full rank (invertible) matrix Λ. This matrix and its inverse will be used directly in the solution to the compound inverse problem. The matrix Λ, or more precisely the inverse of it, can be generated recursively utilizing the principle of inclusion-exclusion

$$P(\bigcup_{i=1}^{N} D_i) = \sum_{i=1}^{N} \left((-1)^{i-1} \sum_{I\subset\{1,\ldots,N\},|I|=i} P(D_I) \right). \tag{6.59}$$

The inner sum runs over all subsets I constructed by the indices $1,\ldots,N$, with

cardinality of I identically i elements. Also above we have defined

$$D_I := \bigcap_{i \in I} D_i. \tag{6.60}$$

Figure 6.4 shows an algorithmically generated example where the fractal structure is again evident. Finally the relation between the inverses of ζ-matrix and Λ matrix can be written as

$$[\Lambda^{-1}(\Lambda^{-1})^T]_{i,j} = \begin{cases} [(\zeta^{-1})^T \zeta^{-1}]_{i+1,j+1}, & \text{when } 1 \leq i,j \leq 2^N - 1 \\ [(\zeta^{-1})^T \zeta^{-1}]_{i+1,j+1} - 1, & \text{if } i = j = 2^N - 1. \end{cases} , \tag{6.61}$$

which is evident by careful inspection of the definitions. Because our space $\mathbb{F}_2^N$ is linear, all subspaces $\mathbb{F}_2^M$ with $1 \leq M < N$ are naturally included in it as $\mathbb{F}_2^M \subseteq \mathbb{F}_2^N$, and we used that property. An additional funny remark is that the probability to have an $n \times n$ matrix over $\mathbb{F}_2$ with determinant $\neq 0$ is $1/4$. This is easily proven.

6.4 Measurements

Regarding proper observables and measurements, we could start to write down suitable theory formulations from which one could perhaps derive semi-analytical distributions for the Bernoulli observables, at least with toy models. However, here our abstraction level is different. This construction is essentially a new model-independent definition of high energy diffraction: different correlations over space-time observables, in terms of rapidity and transverse momentum (and multiplicity), are embedded in the construction and predictions can be obtained directly with any Monte Carlo event generator. We rely only on the observable fiducial final state information. Incidentally, our construction has some similarities with the coherence definition by Glauber in quantum optics [167].

Partial cross sections

To be concrete, a practical measurement procedure of the combinatorial partial cross sections, which may be called also vector fiducial cross sections, goes as follows.

1. Input: Use minimum bias or zero-bias (bunch cross-over) triggered data.
2. Beam Background Filter: Filter event by event the beam-gas and satellite interactions via detector time-domain cuts, if possible. Tune the beam background cuts using a combination of Monte Carlo (pure beam-beam) and data in a way that beam-beam interaction efficiency is preserved.
3. Fiducial Cut Definition: Apply fiducial cuts for pseudorapidity η and transverse momentum p_t. A single fixed set of cuts gives $2^N - 1$ non-zero observable binary combinations. A sliding threshold cuts give a 'trajectory of observables'.
4. Residual Beam Background Correction: Correct residual beam-gas background at statistical level using beam-empty, empty-beam, empty-empty trigger masks: $N \leftarrow N - \sum_j \alpha_j N_j$, where weight factors α_j are obtained from the j-th trigger mask statistics with pre-determined and/or random trigger downscaling and N is the number of events.
5. Pile-Up: Correct pile-up via statistical Möbius inversion using Eq. 6.79, if the run was with high instantaneous luminosity.
6. Luminosity: Map event counts to visible cross section units via integrated luminosity, calibrated via van der Meer scans, see Appendix C.5.
7. Multidimensional Unfolding: Unfold visible partial cross sections to fiducial partial cross sections. One may use 'standard' unfolding algorithms.
8. Output: A vector $\boldsymbol{\sigma}$ (set) of fiducial cross sections.

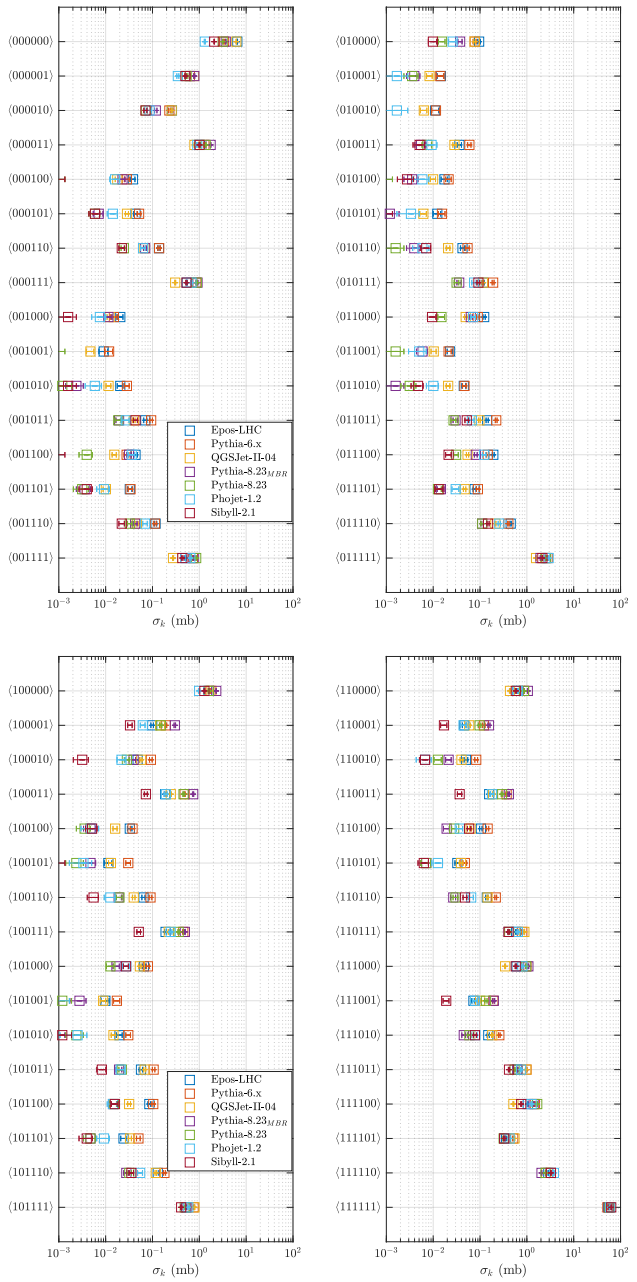

Figure 6.5: Partial cross sections evaluated with proton-proton minimum bias Monte Carlo event generators at $\sqrt{s} = 13$ TeV. Statistical uncertainties are indicated with error bars. The total inelastic cross section for all models is set to 80 mb.

In addition, one may now use the unfolded partial cross sections to obtain highly constraining multidimensional fit extractions of diffractive cross sections and the Pomeron parameters or the maximum diffractive system mass limit, sometimes called the 'coherence limit' [168]. Also, re-projections of other observables are possible. For highly efficient algorithms, see Appendix C.6 and C.7.

In Figure 6.5 we have several Monte Carlo model [163, 169–174] fiducial proton-proton cross sections at $\sqrt{s} = 13$ TeV with cuts suitable for the ALICE experiment. The fiducial acceptance domains for charged particles with $N = 6$ are

$$\eta \in [-7.0, -4.9], [-3.7, -1.7], [-2.0, 0], [0, 2.0], [2.8, 5.1], [4.8, 6.3] \ \wedge \ p_t > 0.05 \text{ GeV}, \tag{6.62}$$

motivated by the nominal acceptances of AD, V0 and SPD detectors of the ALICE experiment. A large variation between models is seen, representing different tunes and design choices. In certain sense, the vector fiducial cross sections are closer to classic diffraction measurements (aperture + screen), than the typical rapidity gap counting. Regarding the multidimensional unfolding, we recommend using several reference MC models together with the detector simulation. One should also characterize the unfolding regularization parameter strength in a data-driven way, for example, using the Shannon information entropy

$$S = -\sum_i P_i \ln P_i, \tag{6.63}$$

where $P_i \equiv \sigma_i / \sum_j \sigma_j$, by calculating the entropy of the unfolded spectrum and the relative entropy between the unfolded and raw combination rates

$$D = \sum_i P_i \ln \left(\frac{P_i}{\tilde{P}_i} \right). \tag{6.64}$$

Naturally, pure simulation studies should be done also. The non-trivial systematics behind efficiency corrections are made here very explicit by the multidimensional unfolding, which accounts for both efficiency losses and the flow of event topologies from one combination to another. This flow is not just because of efficiency losses but also due to major material re-scattering and other detector effects, often hardly understood with single dimensional rapidity gap counting measurements.

To this end, we point out that in principle one could extend the fiducial measurement vector space from the GF(2) finite field $\mathbb{F}_2^N$ to $\mathbb{F}_q^N$, where q is an integer power of a prime number. The prime power is a finite field requirement. We see that this could be useful regarding different rapidity slices containing different extreme (low or high) multiplicity densities in the same event, not necessarily hard scale or jets driven, for example.

Fractal marginal distributions

We proceed as before, but now measure all single dimensional distributions available per binary combination, such as the multiplicity $P(N_{ch})$ and transverse momentum $P(p_t)$ distributions, once the slicing is done over (pseudo)rapidity. For example, assume $N = 3$ slices over pseudorapidity. For the combination $\langle 0, 0, 1\rangle$ we have one non-zero component $P(N_{ch})_{\langle 0,0,[1]\rangle}$, for $\langle 0, 1, 1\rangle$ we get $P(N_{ch})_{\langle 0,1,[1]\rangle}$, $P(N_{ch})_{\langle 0,[1],1\rangle}$ and similarly for transverse momentum. As an illustrative example, we tabulate those in Tables 6.2 and 6.3 with $N = 3$. The number $a(n)$ of non-zero distributions per vector combination is enumerated with $a(0) = 0, a(2n) = a(n), a(2n+1) = a(n)+1$, producing a fractal sequence $0, 1, 1, 2, 1, 2, 2, 3, \ldots$. That is simply the Hamming weight. The total number of distributions is $N2^{N-1}$, the number of edges in an N-hypercube.

1	$\emptyset$	$\emptyset$	$f(\mathcal{O}_x)_{\langle 0,0,[1]\rangle}$
2	$\emptyset$	$f(\mathcal{O}_x)_{\langle 0,[1],0\rangle}$	$\emptyset$
3	$\emptyset$	$f(\mathcal{O}_x)_{\langle 0,[1],1\rangle}$	$f(\mathcal{O}_x)_{\langle 0,1,[1]\rangle}$
$\vdots$		$\vdots$	
8	$f(\mathcal{O}_x)_{\langle [1],1,1\rangle}$	$f(\mathcal{O}_x)_{\langle 1,[1],1\rangle}$	$f(\mathcal{O}_x)_{\langle 1,1,[1]\rangle}$
$\Sigma\downarrow$	$\underbrace{f(\mathcal{O}_x)_{\langle [1],-,-\rangle}}_{\int_{\Delta\eta_1} d\eta}$	$\underbrace{f(\mathcal{O}_x)_{\langle -,[1],-\rangle}}_{\int_{\Delta\eta_2} d\eta}$	$\underbrace{f(\mathcal{O}_x)_{\langle -,-,[1]\rangle}}_{\int_{\Delta\eta_3} d\eta}$

Table 6.2: Combinatorially embedded marginal 1D-distributions of the observable $\mathcal{O}_x$ with $N = 3$ slices over pseudorapidity.

1	$\emptyset$	$\emptyset$	$f(\mathcal{O}_x, \mathcal{O}_y)_{\langle 0,0,[1]\rangle}$
2	$\emptyset$	$f(\mathcal{O}_x, \mathcal{O}_y)_{\langle 0,[1],0\rangle}$	$\emptyset$
3	$\emptyset$	$f(\mathcal{O}_x, \mathcal{O}_y)_{\langle 0,[1],1\rangle}$	$f(\mathcal{O}_x, \mathcal{O}_y)_{\langle 0,1,[1]\rangle}$
$\vdots$		$\vdots$	
8	$f(\mathcal{O}_x, \mathcal{O}_y)_{\langle [1],1,1\rangle}$	$f(\mathcal{O}_x, \mathcal{O}_y)_{\langle 1,[1],1\rangle}$	$f(\mathcal{O}_x, \mathcal{O}_y)_{\langle 1,1,[1]\rangle}$
$\Sigma\downarrow$	$\underbrace{f(\mathcal{O}_x, \mathcal{O}_y)_{\langle [1],-,-\rangle}}_{\int_{\Delta\eta_1} d\eta}$	$\underbrace{f(\mathcal{O}_x, \mathcal{O}_y)_{\langle -,[1],-\rangle}}_{\int_{\Delta\eta_2} d\eta}$	$\underbrace{f(\mathcal{O}_x, \mathcal{O}_y)_{\langle -,-,[1]\rangle}}_{\int_{\Delta\eta_3} d\eta}$

Table 6.3: Combinatorially embedded marginal 2D-distributions of observables $(\mathcal{O}_x, \mathcal{O}_y)$ with $N = 3$ slices over pseudorapidity.

We emphasize here that this is just cut-and-count with bookkeeping, a well defined fiducial measurement. The problem of unfolding from detector level to particle level, however, can be non-trivial. We believe that these fractal distributions together with combinatorial partial cross sections can be key handles to test the AGK rules and (topological) fluctuations, perhaps driven by anomalous QCD color field configurations. A potentially related theory discussion can be found in [175, 176].

Threshold gapflow trajectories

The idea here is a straightforward one. One measures $2^N - 1$ fiducial vector cross sections over rapidity as a function of a threshold condition. The threshold condition can be global over rapidity, or local. For example, a minimum transverse momentum threshold, a minimum multiplicity threshold or a minimum flavor content threshold (e.g. strangeness).

The measurement is illustrated in Figure 6.6, where the transverse momentum threshold is changed for four different average particle densities, particles drawn from a simple Poisson distribution with the given mean value. Here we used a simple soft exponential p_t^2 distribution with particles generated uniformly over rapidity with $N = 3$ slices. The exponential in p_t^2 results in a Rayleigh type p_t-distribution with Gaussian distributed p_x and p_y components. Now it is very easy to see that rapidity gaps are not invariant under transverse momentum cutoffs, a well known fact but sometimes a forgotten truth. Depending on the particle density and the cutoff, different event topology rates may have a cross over. The transition to pQCD with power law tails will be interesting.

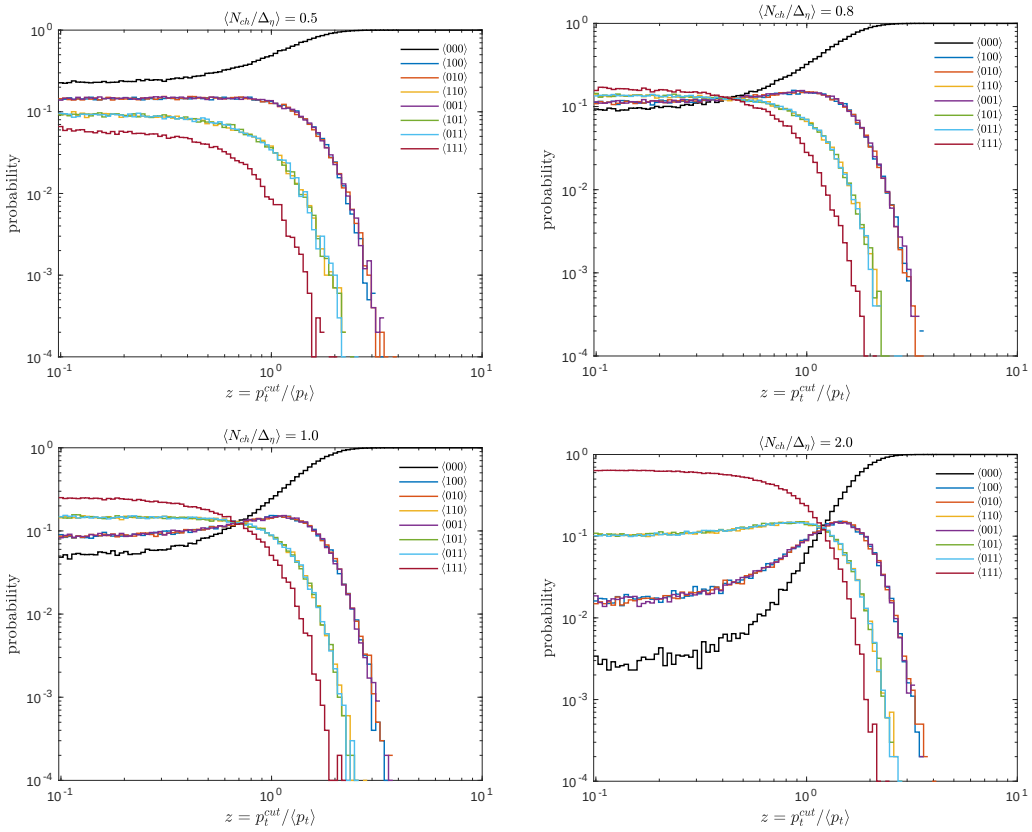

Figure 6.6: A toy simulation of 'gapflow trajectories' as a function of the normalized transverse momentum cutoff $z = p_t^{cut}/\langle p_t \rangle$ with four different particle density.

6.5 Combinatorial supercompound Poisson process

By using the tools introduced earlier, we formulate a Poisson superposition problem and its inverse. This is essentially a minimal non-trivial mathematical model needed to understand the problematics involved, before trying to deconstruct more complicated scenarios. This model has no interfering amplitude structures, *a priori*, whereas the AGK type compound interactions do contain interfering amplitudes, by construction. We will see later that certain boundary conditions will force the solutions to this problem towards more complicated scenarios.

We start by assuming a Poisson fluctuating random variable $K \sim \frac{\mu^k}{k!}e^{-\mu}$ with mean μ and argument $k \in \mathbb{N}$ denoting the number of simultaneous interactions, such as multiparton interactions or multiple independent proton-proton interactions. Both cases are treated analogously, modulo the energy-momentum conservation differences. The probability generating function for Poisson is $G(z) = \mathbb{E}(z^X) = \exp(\mu(z-1))$ which is useful for deriving the compound moments. In general, the expectation values and variances of a random variable $C_{|K>0} = \sum_{j=1}^{K} X_j$ after compounding stochastics are obtained from the compound relations

$$\langle C \rangle = \mathbb{E}_{K>0}[\mathbb{E}_{C|K>0}(C)] = \langle K > 0 \rangle \langle X \rangle \tag{6.65}$$

$$\begin{aligned} \mathrm{Var}[C] &= \mathbb{E}_{K>0}[\mathrm{Var}_{C|K>0}(C)] + \mathrm{Var}_{K>0}[\mathbb{E}_{C|K>0}(C)] \\ &= \langle K > 0 \rangle \mathrm{Var}[X] + \mathrm{Var}[K > 0](\langle X \rangle)^2. \end{aligned} \tag{6.66}$$

The zero-truncated Poisson mean and variance of K are given in Appendix C.4. We need the zero truncated distribution, because the 0 case is not a physical observable. However, all the N-point moments necessary to describe the multidimensional case, i.e. the case $N > 1$, are *not* handled with these simple formulas but only via the full machinery which follows.

For preliminaries, the ordinary binomial coefficient is

$$C_k^n = \binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 0 \leq k \leq n \tag{6.67}$$

which describes how many ways k numbers can be chosen from a set of n numbers. Counting Bernoulli random variables in a binomial setup, one obtains the Binomial distribution. By counting random variables in a multivariate Bernoulli setup, we obtain analogously the multivariate Binomial distribution. However, for the sake of practicality we use now the multinomial distribution with random variables $X_1, X_2, \ldots, X_{2^N-1}$, which is as many as we have non-zero fiducial vectors in

the binary space. The multinomial probability distribution is

$$P_m(X_1 = x_1, \ldots, X_{2^N-1} = x_{2^N-1}|\mathbf{p}) = \frac{k!}{\prod_{c=1} x_c!} \prod_{c=1}^{2^N-1} p_c^{x_c} \tag{6.68}$$

$$\text{with} \quad \sum_{c=1}^{2^N-1} x_c = k, \quad x_c \in \mathbb{N} \ \text{ and } \ \sum_c p_c = 1, \tag{6.69}$$

which represents sampling or occupying a different number x_i of different fiducial vectors per mixed event, the number of vectors in superposition per event is the number k. The first term is the multinomial coefficient easily derived using the binomial coefficient, also known in Maxwell-Boltzmann statistics. The probability generating function is here given by $G(\mathbf{z}) = \left(\sum_{c=1}^{2^N-1} p_c z_c\right)^k$, $\mathbf{z} \in \mathbb{C}^n$ which is useful, when expanded, in interpreting multinomial distribution as polynomial coefficients p_c with unit sum.

Here *distinguishability* means that we can separate between different final state vectors but not between a vector of the same type. This is a sampling with replacement scheme. Sampling without replacement gives, for example, a multivariate hypergeometric distribution instead of the multinomial. Similar non-multinomial sampling scenarios are encountered with Fermi-Dirac and Bose-Einstein statistics, for example.

Direct model

We define a generative direct model $f : \Theta \to Y$, where Θ denotes the space of parameters and Y the space of observables. The unknown parameters are in $\mathbf{p} = [p_1, p_2, \ldots, p_n]^T \in \Theta$ and the compounding process diluted or enhanced combination rate measurements are in $\mathbf{y} = [y_1, y_2, \ldots, y_n]^T \in Y$. The model estimates for the observed rates y_c are generated, as a sum over product of combinatorially selected multinomial distribution terms W_{ck} and Poisson probabilities $P_p(k = j)$ order by order as

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \frac{1}{1 - e^{-\mu}} \begin{pmatrix} W_{11} & W_{12} & W_{13} \ldots W_{1\infty} \\ W_{21} & W_{22} & W_{23} \ldots W_{2\infty} \\ & \vdots & \ddots \\ W_{n1} & W_{n2} & W_{n3} \ldots W_{n\infty} \end{pmatrix} \begin{pmatrix} P_p(k = 1; \mu) \\ P_p(k = 2; \mu) \\ P_p(k = 3; \mu) \\ \vdots \\ P_p(k = \infty; \mu) \end{pmatrix}, \tag{6.70}$$

where we have an infinite series. The (re)-normalization factor $P_p(0;\mu) = 1 - e^{-\mu}$ is needed, because we exclude (truncate) the null case $k = 0$ of the Poisson density. Solving the inverse of Eq. 6.70 is a non-linear problem, because the matrix elements of W depend on the unknown parameters. Now more explicitly written, the *compound Poisson* model equation for each probability component is

$$\begin{aligned} y_c &= \frac{1}{1-e^{-\mu}} \sum_{k=1}^{\infty} \frac{\mu^k}{k!} e^{-\mu} W_{ck} \\ &= \frac{e^{-\mu}}{1-e^{-\mu}} \sum_{k=1}^{\infty} \frac{\mu^k}{k!} \left\{ \sum_{\{x_j\} \subset \Omega_{ck}} \frac{k!}{\prod_{j=1}^{n} x_j!} \prod_{j=1}^{n} p_j^{x_j} \right\}, \end{aligned} \tag{6.71}$$

with $\sum_c y_c = 1$. The multinomial term and its values of x_j are evaluated over all valid combinations or partitions for y_c from the set of n-tuples Ω_{ck}, that is, those which are allowed by pileup poset combinatorics. Formally, this set of compositions can be enumerated with

$$\Omega_{ck} = \left\{ (x_1, \ldots, x_j, \ldots, x_n) \,|\, \bigvee_j x_j \mathbf{b}_j = \mathbf{b}_i \text{ and } \sum_j x_j = k \right\}, \tag{6.72}$$

where the boolean $\bigvee$ operator takes care of 'summing' the binary vectors $\mathbf{b}_j$ of multiplicity x_j and thus evaluating the pileup compositions. The p_j are the probabilities we want to find out by inverting the compound Poisson process, that is, the probabilities of each $n = 2^N - 1$ fiducial final state.

Example $N = 2$: a (heavily) truncated Poisson series at $\mathcal{O}(\mu^2)$ is

$$W_{3\times 2} = \begin{pmatrix} p_1 & P_m(2,0,0) \\ p_2 & P_m(0,2,0) \\ p_3 & P_m(1,1,0) + P_m(1,0,1) + P_m(0,1,1) + P_m(0,0,2) \end{pmatrix}. \tag{6.73}$$

Now a brute force inverse $f^{-1} : Y \to \Theta$ would be to use Eq. 6.71 and then estimate parameters by minimizing a non-linear χ^2-function between the generated $\hat{y}_c$ and measured y_c values $\hat{\mathbf{p}} = \arg\min_{\mathbf{p}} \chi^2$, where $\chi^2 = (\hat{\mathbf{y}} - \mathbf{y})^T \Sigma_{\mathbf{y}}^{-1} (\hat{\mathbf{y}} - \mathbf{y})$ and usually $\Sigma_{\mathbf{y}}^{-1} = \mathrm{diag}(1/\delta y_c^2)$. This is a typical fit approach, which is the statistical Maximum Likelihood (ML) solution when the measurement noise (counting fluctuations) is approximated with Gaussian distribution. A ML solution taking into account the multinomial counting fluctuations can be formulated also. However, if the direct model is constructed just order by order in k, this approach is computationally

extremely expensive at high values of k (and N) due to factorial growth. This can be seen in Table C.2 in Appendix C.2. Instead of using this, we will formulate an 'all-order' solution based on the Möbius inversion.

Solution for $N \leq 2$

The first case $N = 1$ is trivial, because we have just one Bernoulli observable, no combinations involved and because of our normalization condition $\sum_c p_c = 1$. Thus, the most interesting quantity is directly the Poisson distribution μ-value and probabilities for $k = 0, 1, 2, \ldots$.

The case $N = 2$ is straightforward. An explicit formula for y_1 is obtained by noticing that the set Ω_{1k} allows only 'autocompound' for all k. Thus $B_1 \equiv k$ and $B_2 \equiv B_3 \equiv 0$ for all values of k, which gives

$$\begin{aligned} y_1 &= \frac{e^{-\mu}}{1-e^{-\mu}} \sum_{k=1}^{\infty} \frac{\mu^k}{k!} \left(\frac{k!}{k!0!0!} p_1^k p_2^0 p_3^0 \right) \\ &= \frac{e^{-\mu}}{1-e^{-\mu}} \sum_{k=1}^{\infty} \frac{\mu^k}{k!} p_1^k \\ &= \frac{e^{\mu p_1} - 1}{e^{\mu} - 1}. \end{aligned} \tag{6.74}$$

Above we used the definition of Taylor series for e^x. By calculating similarly for y_2 by replacing above p_1 with p_2, then we get y_3 using the conservation of probability

$$y_3 = 1 - \sum_{c=1}^{2} \frac{e^{\mu p_c} - 1}{e^{\mu} - 1}. \tag{6.75}$$

Now the inverse is given by

$$\hat{p}_c = \frac{\ln((e^{\mu} - 1)y_c + 1)}{\mu}, \quad \text{for } c = 1, 2 \tag{6.76}$$

and again by conservation of probability

$$\hat{p}_3 = 1 - \sum_{c=1}^{2} \frac{\ln((e^{\mu} - 1)y_c + 1)}{\mu}, \tag{6.77}$$

which completes the solution.

Solution for $N \geq 1$

The arbitrary N case follows recursively from the $N = 2$ case and can be done in practice by constructing the matrix Λ and its inverse according to the incidence algebra described earlier. The main result follows. The combinatorial statistics is

$$\boxed{\mathbf{y}(\mathbf{p};\mu) = \Lambda^{-1}\frac{\exp[-\mu\Lambda\mathbf{p}]-1}{e^{-\mu}-1}} \tag{6.78}$$

which has also the inverse we are looking for

$$\boxed{\hat{\mathbf{p}}(\mathbf{y};\mu) = \Lambda^{-1}\frac{\ln\left[(e^{-\mu}-1)\Lambda\mathbf{y}+1\right]}{-\mu}.} \tag{6.79}$$

The exponential and logarithm are taken vector entry wise, such that the Taylor series representation is using the Hadamard power: $\exp[\mathbf{v}] = \sum_{k=0}^{\infty}\frac{1}{k!}\mathbf{v}^{\odot k}$. Equations 6.78, 6.79 are relying on the matrix Λ constructing linear combinations of the probabilities of elements in $\mathbb{F}_2^N$ and the property that multinomial probability distribution is factorized as a product of different terms. The multinomial distribution obeys the so-called grouping property: re-partitioning conserves the multinomial distribution. Similarly Λ^{-1} deconstructs the linear combinations, and in between there is a non-linear transformation according to the compound Poisson statistics. One can understand this also in terms of convolution theorem which turns the convolution into a usual product in the transform (Fourier) domain or vice verse. Here the Möbius inversion has the same role as Fourier decomposition.

The example with $N = 3$ gives a vector

$$\hat{\mathbf{p}} = \frac{1}{\mu}\begin{pmatrix} \ln(e_-^{\mu}y_1+1) \\ \ln(e_-^{\mu}y_2+1) \\ -\sum_{c=1,2}\ln(e_-^{\mu}y_c+1)+\ln(1+\sum_{c=1,2,3}e_-^{\mu}y_c) \\ \ln(e_-^{\mu}y_4+1) \\ -\sum_{c=1,4}\ln(e_-^{\mu}y_c+1)+\ln(1+\sum_{c=1,4,5}e_-^{\mu}y_c) \\ -\sum_{c=2,4}\ln(e_-^{\mu}y_c+1)+\ln(1+\sum_{c=2,4,6}e_-^{\mu}y_c) \\ \mu+\sum_{c=1,2,4}\ln(e_-^{\mu}y_c+1)-\ln(1+\sum_{c=1,2,3}e_-^{\mu}y_c)-\ln(1+\sum_{c=1,4,5}e_-^{\mu}y_c)-\ln(1+\sum_{c=2,4,6}e_-^{\mu}y_c) \end{pmatrix},$$

where by conservation of probability we chose to fix $y_7 = 1-\sum_{c=1}^{6}y_c$ and for compacting the notation we set $e_-^{\mu} \equiv e^{\mu}-1$. The fractal structure of the problem is visible again. Components $c = 1, 2, 4$ only undergo autocompound, thus no mixing

between other vectors. These are the unit basis vectors $[1, 0, \ldots, 0]^T, [0, 1, \ldots, 0]^T$ and $[0, 0, \ldots, 1]^T$ and cannot be thus constructed as a linear combination of other vectors. A practical remark when working with non-linear equations with exponential and logarithmic functions: numerical evaluations are prone to floating point accuracy problems at high μ values.

Maximum Input Entropy

The constraints used in the construction required that both $\mathbf{p}$ and $\mathbf{y}$ are probability distributions and that the problem satisfies the combinatorial incidence algebra, together with the multinomial statistics embedded in the Poisson process. Next, we treat certain special solutions.

One interesting case to study is the maximum input entropy case $\mathbf{p} \equiv \mathbf{1}/n$, that is, all different combinations are equally probably. We go through the case $N = 3$ here explicitly. First we solve

$$\begin{aligned} \ln(e_-^\mu y_1 + 1)/\mu &= \frac{1}{n} = \frac{1}{7} \\ y_1 &= \frac{e^{\mu/7} - 1}{e_-^\mu} \equiv y_2 \equiv y_4. \end{aligned} \tag{6.80}$$

Then get we the third component by first substituting y_1

$$\begin{aligned} [-2\frac{\mu}{7} + \ln(1 + 2(e^{\mu/7} - 1) + e_-^\mu y_3)]/\mu &= \frac{1}{7} \\ y_3 &= \frac{-2e^{\mu/7} + e^{3\mu/7} + 1}{e_-^\mu} \equiv y_5 \equiv y_6. \end{aligned} \tag{6.81}$$

The last component is obtained by conservation of probability

$$y_7 = 1 - y_{1:6} = \frac{3e^{\mu/7}(1 - e^{2\mu/7})}{e_-^\mu} + 1, \tag{6.82}$$

where $y_{1:6}$ denotes a sum over all the given indices. It is interesting also to take a look at

$$f(\mu) = \sum_{c=3,5,6} y_c - \sum_{c=1,2,4} y_c = \frac{3(2 - 3e^{\mu/7} + e^{3\mu/7})}{e_-^\mu}. \tag{6.83}$$

Maximum input entropy solutions are represented in Figure 6.7 as a function of μ. We see that there is a fixed maximum output entropy point when $\mathbf{y} = \mathbf{1}/n$ with the critical μ-value depending on N.

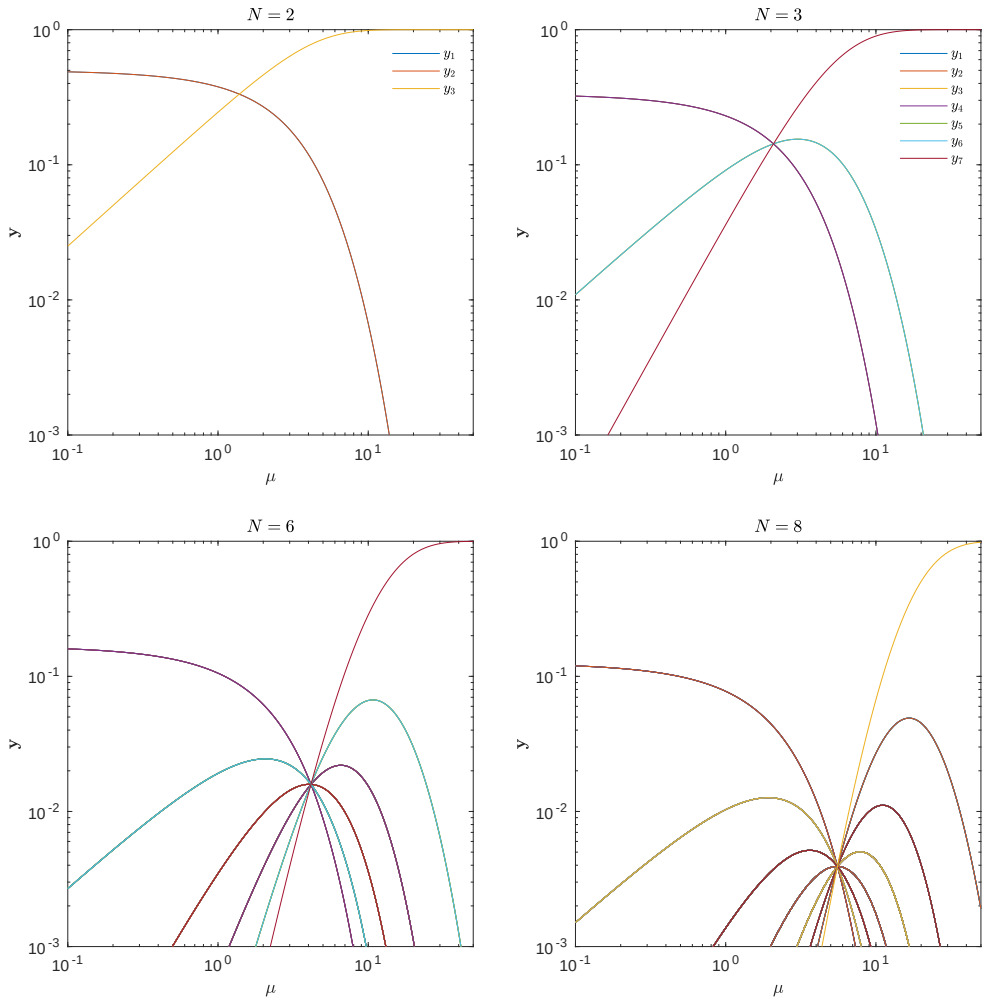

Figure 6.7: Trajectories of $\mathbf{y}$ with the maximum input entropy initial condition $\mathbf{p} \equiv \mathbf{1}/n$.

Maximum output entropy

Here we construct solutions with maximum output entropy. Examples of functional behavior are given in Figure 6.8. We see that it is not possible to obey the constraints of all probabilities being non-negative, thus we obtain $\mathbf{p}$ with negative elements when μ is high enough. An interesting oscillating phenomenon appears with nodes at higher μ due to the polynomial structure of our construction.

Negative values are seemingly unphysical unless one extends the domain of the problem to include also 'negative probabilities' – sometimes called *quasi-probability distributions*, utilized first by Wigner [177]. In Wigner's construction, these are fundamentally non-classical distributions of quantum states. The construction here has certain analogous behavior, essentially because it is the vector $\mathbf{y}$ which are observables and the vector $\mathbf{p}$ contains the parameters of the states, which may have non-classical distributions.

Non-classical distributions of $\mathbf{p}$ here may thus be useful for quantum interference, absorption, shadowing and diffraction. A natural extension is complex vector distributions. Negative probabilities as a useful intermediate step of calculations were also discussed in a well-known article by Feynman [178] in the context of two non-commuting observables.

Master Equation

The first order derivative of $\mathbf{y}$ with respect to μ results in

$$\frac{\partial \mathbf{y}}{\partial \mu} = \Lambda^{-1} \left(\frac{\exp[-\mu \Lambda \mathbf{p}] - 1}{e^{\mu}(e^{-\mu} - 1)^2} - \frac{\Lambda \mathbf{p} \odot \exp[-\mu \Lambda \mathbf{p}]}{e^{-\mu} - 1} \right). \tag{6.84}$$

The higher order derivatives follow similarly, with lengthy expressions which we evaluated with symbolic computer algebra. It is this expression which can be called the *master equation* of describing the evolution of probabilities in terms of the μ-value. Visualized, we obtain wavelet like behavior shown in Figure 6.9. Now obvious future direction would be to probe a set of stochastic equations which would use these basic building blocks. In Figure 6.9 we show also the 'analytically continued' case with negative μ-values for completeness and leave the possible applications out of discussion here. The values of $\mathbf{y}$ are positive with positive μ values, but obtain oscillating positive and negative values at negative μ. The behavior is actually opposite, if we invert the case and study values of $\mathbf{p}$ at negative μ when $\mathbf{y} \equiv \mathbf{1}/n$.

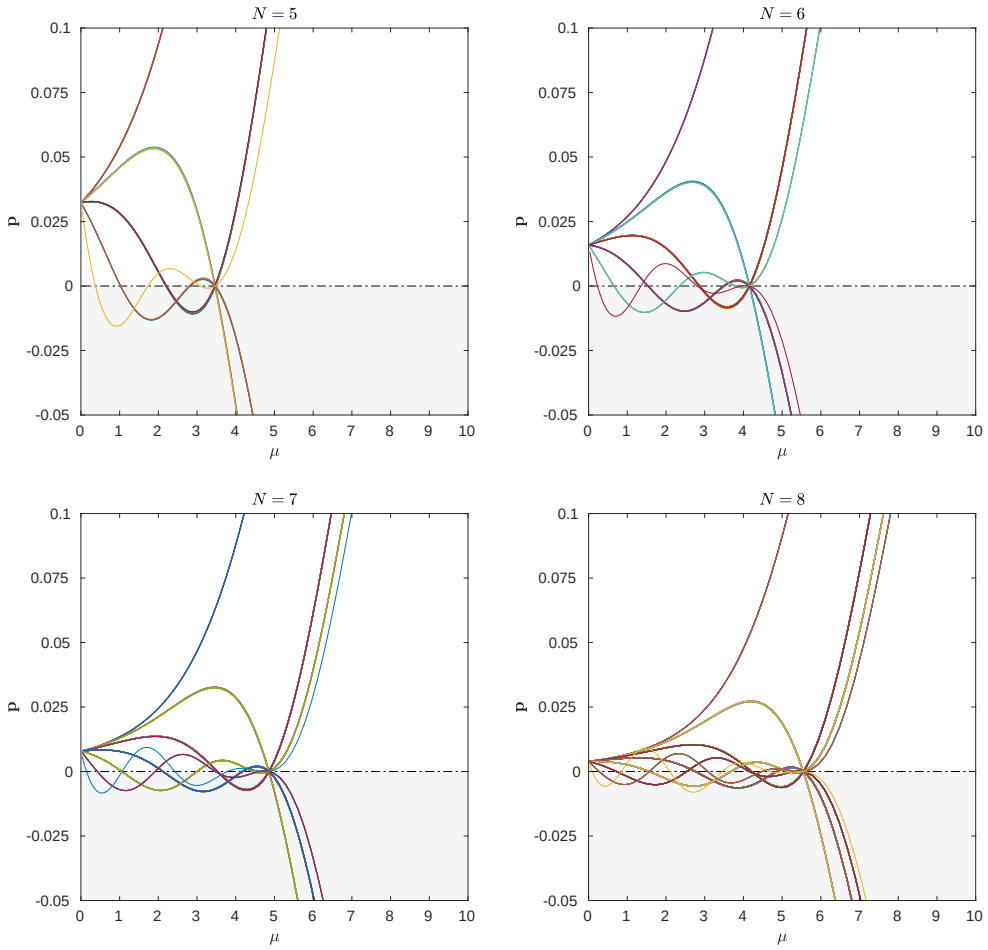

Figure 6.8: Inverted trajectories of $\mathbf{p}$ with the maximum output entropy $\mathbf{y} \equiv \mathbf{1}/n$ boundary condition. The gray area shows the non-positive definite domain.

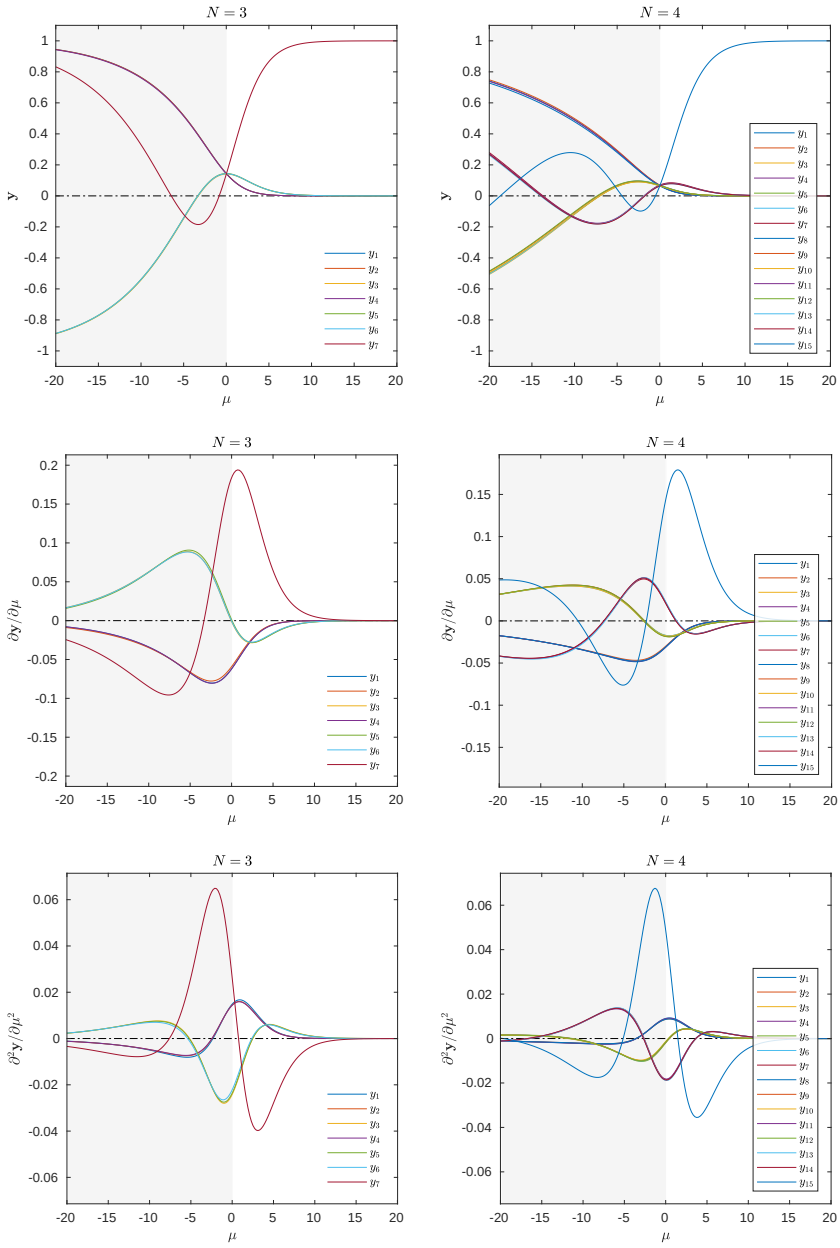

Figure 6.9: Trajectories (1st row) and first and second order derivatives (2nd and 3rd rows) with the $\mathbf{p} \equiv \mathbf{1}/n$ input, for $N = 3$ and $N = 4$. We have added a tiny perturbation to the input for visualization (parity degeneracy).

Semi-Bose-Einstein construction

The construction above was the most relaxed one, in a sense that we allow for example $[0,1]+[0,1] \to [0,1]$ type vector combinations. However, one can construct a modified statistics by forbidding same the vector combination appearing more than once, but not limiting occupation number per vector dimension. This will limit a large number of possible combinations meaning all 'autocompound' is forbidden. In even more strict construction, on the other hand, we would allow only one occupation per vector dimension. We recall that different canonical particle statistics are most easily derived using the combinatorial method presented in any advanced book on statistical mechanics.

The case $N = 2$ is

$$\begin{aligned} y_1 &= P_{K=1} P_m(1,0,0) \\ y_2 &= P_{K=1} P_m(0,1,0) \\ y_3 &= P_{K=1} P_m(0,0,1) \\ &\quad + P_{K=2}[P_m(1,1,0) + P_m(1,0,1) + P_m(0,1,1)] \\ &\quad + P_{K=3} P_m(1,1,1), \end{aligned} \tag{6.85}$$

where the notation is the same as before.
The case $N = 3$ is

$$\begin{aligned} y_1 &= P_{K=1} P_m(1,0,0,0,0,0,0) \\ y_2 &= P_{K=1} P_m(0,1,0,0,0,0,0) \\ y_3 &= P_{K=1} P_m(0,0,1,0,0,0,0) \\ &\quad + P_{K=2}[P_m(1,1,0,0,0,0,0) + P_m(1,0,1,0,0,0,0) + P_m(0,1,1,0,0,0,0)] \\ &\quad + P_{K=3} P_m(1,1,1,0,0,0,0) \\ y_4 &= P_{K=1} P_m(0,0,0,1,0,0,0) \\ y_5 &= P_{K=1} P_m(0,0,0,0,1,0,0) \\ &\quad + P_{K=2}\,[P_m(1,0,0,1,0,0,0) + P_m(1,0,0,0,1,0,0) + P_m(0,0,0,1,1,0,0)] \\ &\quad + P_{K=3} P_m(1,0,0,1,1,0,0) \\ y_6 &= P_{K=1} P_m(0,0,0,0,0,1,0) \\ &\quad + P_{K=2}\,[P_m(0,1,0,1,0,0,0) + P_m(0,0,0,1,1,0,0) + P_m(0,1,0,0,1,0,0)] \\ &\quad + P_{K=3} P_m(0,1,0,1,0,1,0) \\ y_7 &= 1 - y_{1:6}, \end{aligned} \tag{6.86}$$

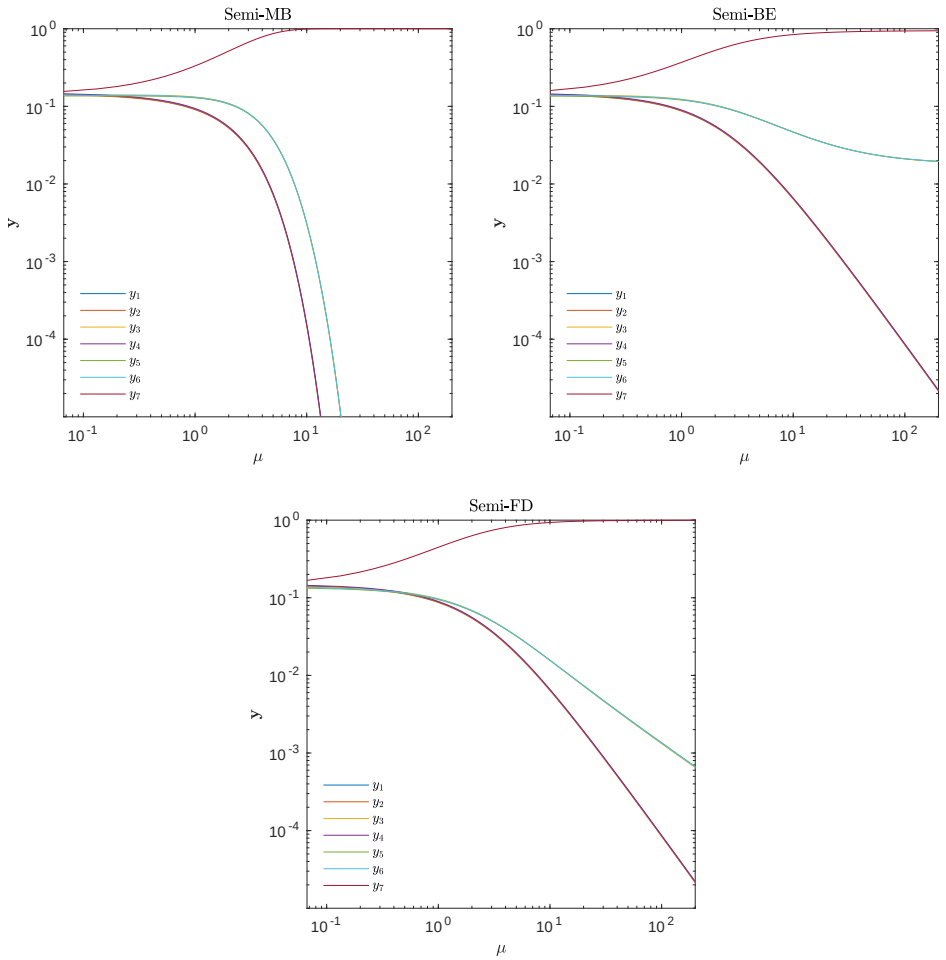

Figure 6.10: Semi-Maxwell-Boltzmann, Semi-Bose-Einstein and Semi-Fermi-Dirac boundary conditions with the maximum input entropy $\mathbf{p} = \mathbf{1}/n$ initial condition.

where the last component is obtained simply by conservation of probability. In this 'Semi-Bosonic' case, we did not find such a simple closed-form formula as the matrix equation 6.79, but solutions for any N can be found simply algorithmically using symbolic computer algebra.

Semi-Fermi-Dirac construction

In the Semi-Fermi-Dirac construction, which is the most strict one we consider here, we allow only one occupation per vector element.

The case $N = 2$ is

$$\begin{aligned} y_1 &= P_{K=1} P_m(1,0,0) \\ y_2 &= P_{K=1} P_m(0,1,0) \\ y_3 &= P_{K=1} P_m(0,0,1) + P_{K=2} P_m(1,1,0). \end{aligned} \tag{6.87}$$

The case $N = 3$ is

$$\begin{aligned} y_1 &= P_{K=1} P_m(1,0,0,0,0,0,0) \\ y_2 &= P_{K=1} P_m(0,1,0,0,0,0,0) \\ y_3 &= P_{K=1} P_m(0,0,1,0,0,0,0) + P_{K=2} P_m(1,1,0,0,0,0,0) \\ y_4 &= P_{K=1} P_m(0,0,0,1,0,0,0) \\ y_5 &= P_{K=1} P_m(0,0,0,0,1,0,0) + P_{K=2} P_m(1,0,0,1,0,0,0) \\ y_6 &= P_{K=1} P_m(0,0,0,0,0,1,0) + P_{K=2} P_m(0,1,0,1,0,0,0) \\ y_7 &= 1 - y_{1:6}. \end{aligned} \tag{6.88}$$

Again, the any N case can be constructed algorithmically.

The results are shown in Figure 6.10, where different constraints on the combinations result in different asymptotic behavior in the tails.

6.6 Invertibility simulations

To evaluate the finite sample statistics invertibility from $\mathbf{y}$ into $\mathbf{p}$ using Equation 6.79, simulations are necessary. Standard numerical uncertainty methods such as bootstrapping may be used to quantify the uncertainties in-situ. We do the simulation in a fully controlled way and use the multidimensional Dirichlet distribution

$$f_D(x_1,\ldots,x_n;\alpha_1,\ldots,\alpha_n) = \frac{1}{B(\boldsymbol{\alpha})}\prod_{i=1}^{n} x_i^{\alpha_i-1}, \text{ where } B(\boldsymbol{\alpha}) = \frac{\prod_{i=1}^{n}\Gamma(\alpha_i)}{\Gamma\left(\sum_{i=1}^{n}\alpha_i\right)} \tag{6.89}$$

as a random distribution for probabilities of $\mathbf{p}$ with a constraint $\sum_c p_c = 1$. This constraint is by construction included in the Dirichlet distribution. The normalization factor is the multivariate Beta function, seemingly closely related to the famous Veneziano amplitudes of early string theory. We illustrate the Dirichlet distribution drawn realizations in Figure 6.13, where we observe how the Shannon entropy $S(\mathbf{y})$ of the superposition result is running as a function of the compounding Poisson process μ-value. Interestingly, the entropy has often a non-monotonic, somewhat universal behavior. Picking up random distributions allows us to sample the spectrum of different probability distributions of $\mathbf{p}$ maximally. This means a wide range of possible applications can be effectively probed, in a model independent way. We set the parameter vector of the Dirichlet distribution $\boldsymbol{\alpha} = \alpha\mathbf{1}$ with a concentration parameter $\alpha \in \{0.5, 1\}$. The case $\alpha = 1$ corresponds to the maximally uniform case in the multidimensional space of probabilities.

As a measure of agreement between the estimated probabilities and the true ones, we use the Kolmogorov-Smirnov type test statistic

$$D = \max|F(i) - \hat{F}(i)| \in [0,1], \quad i = 1,\ldots,n \tag{6.90}$$

where $F(i) = \sum_{j=1}^{i} p_j$ is the cumulative distribution function (CDF) of the true components of $\mathbf{p}$ and $\hat{F}(i)$ is the CDF of the estimate $\hat{\mathbf{p}}$. We are now neglecting all the technicalities related to how rigorously extend the KS test to discrete distributions. We calculate only the test statistic itself, and do not calculate the significance level or p-value. In principle this could be done using the Kolmogorov distribution.

The inversion results of several simulations are shown in Figures 6.11 and 6.12. The dashed lines correspond to uncorrected distributions and the solid ones for inverted results and error bars are 1 sigma (68 CL) uncertainties around the median value, obtained via Monte Carlo runs. The μ-value is varied between 10^{-3} and 30. To demonstrate the $\sqrt{N_E}$ scaling or asymptotic efficiency of the algorithm, we vary the number of simulated events N_E. We observe a clear saturation in the inversion

process at high μ-values, because at that point all event signatures are almost identical. That is, all binary observables are $B_i \equiv 1$ for all $i = 1, \ldots, N$. The distributions for the relative components errors

$$\Delta_c \equiv (\hat{p}_c - p_c)/p_c, \tag{6.91}$$

are shown in Figure 6.12 sampled over different μ-value runs. In the histogram legend, the mean square error $\langle(\hat{p}_c - p_c)^2\rangle$ of the estimates are shown. The estimates are seen to be numerically unbiased, as expected from the construction, thus the mean square error of the estimates is equal to the variance of the estimator. In Figure 6.12 we see also that the relative component error distribution variance is seen to grow as a function of dimension N, demonstrating growing complexity in the combinatorial mixing process. The uncorrected residuals are significantly biased, by the nature of the problem which results in extreme mixing between event topologies. These simplified simulations give also new orthogonal insight into the hardness of understanding soft inclusive QCD processes in detail where mixing may happen at more complicated amplitude level.

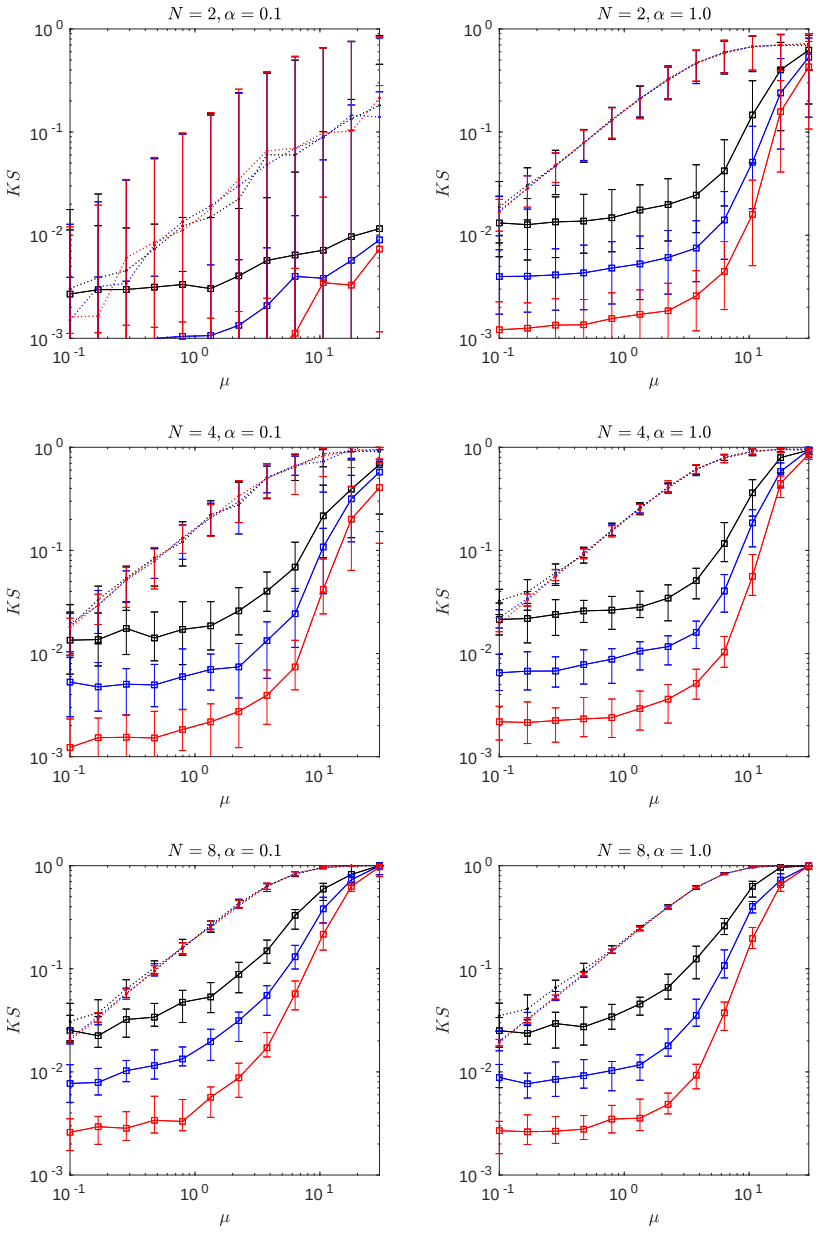

Figure 6.11: Kolmogorov-Smirnov error on the vertical axis. Median values and $\pm 1\sigma$ intervals as a function of μ in different dimensions N. The simulated number of events are $N_E = 10^3, 10^4, 10^5$ counts, denoted with black, blue and red markers. Solid lines are after, and dashed lines before the inversion.

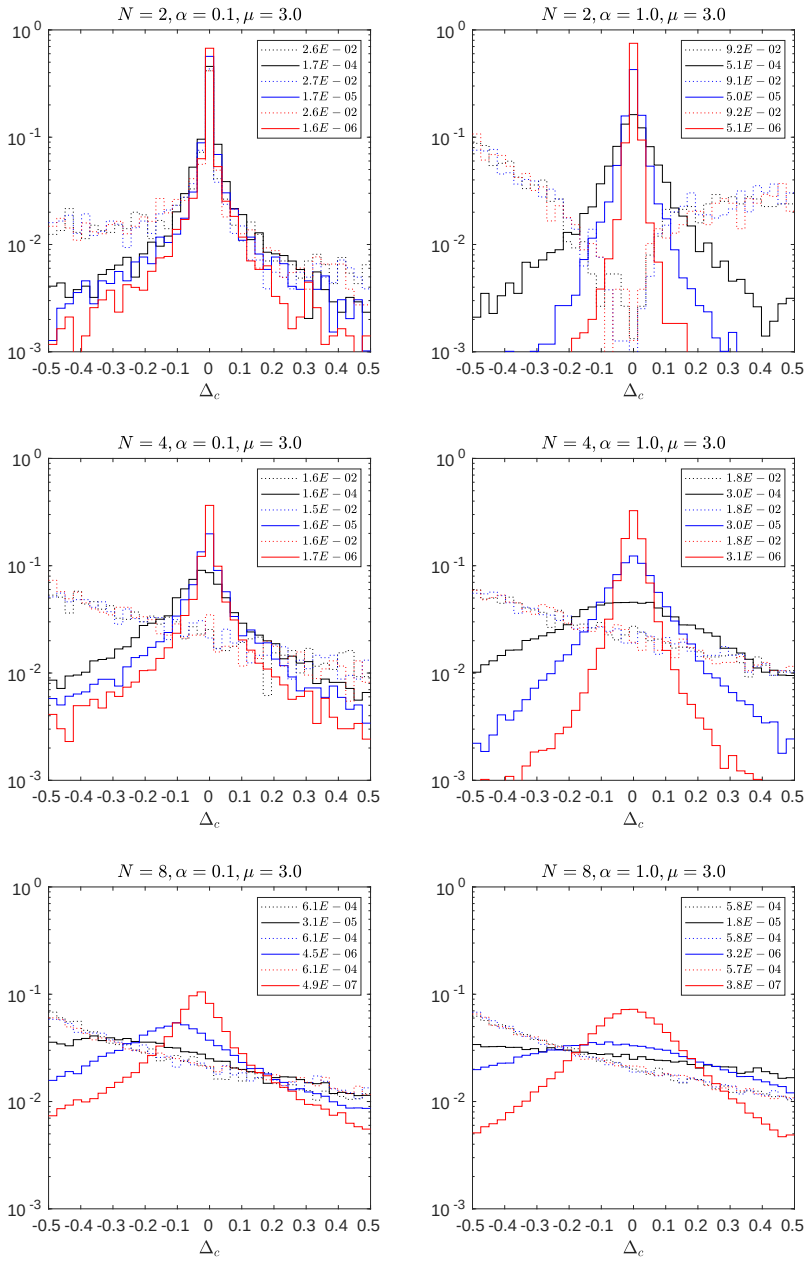

Figure 6.12: Relative component error Δ_c distributions in different dimensions N. The simulated number of events are $N_E = 10^3, 10^4, 10^5$ counts, denoted with black, blue and red markers. Solid lines are after, and dashed lines before the inversion.

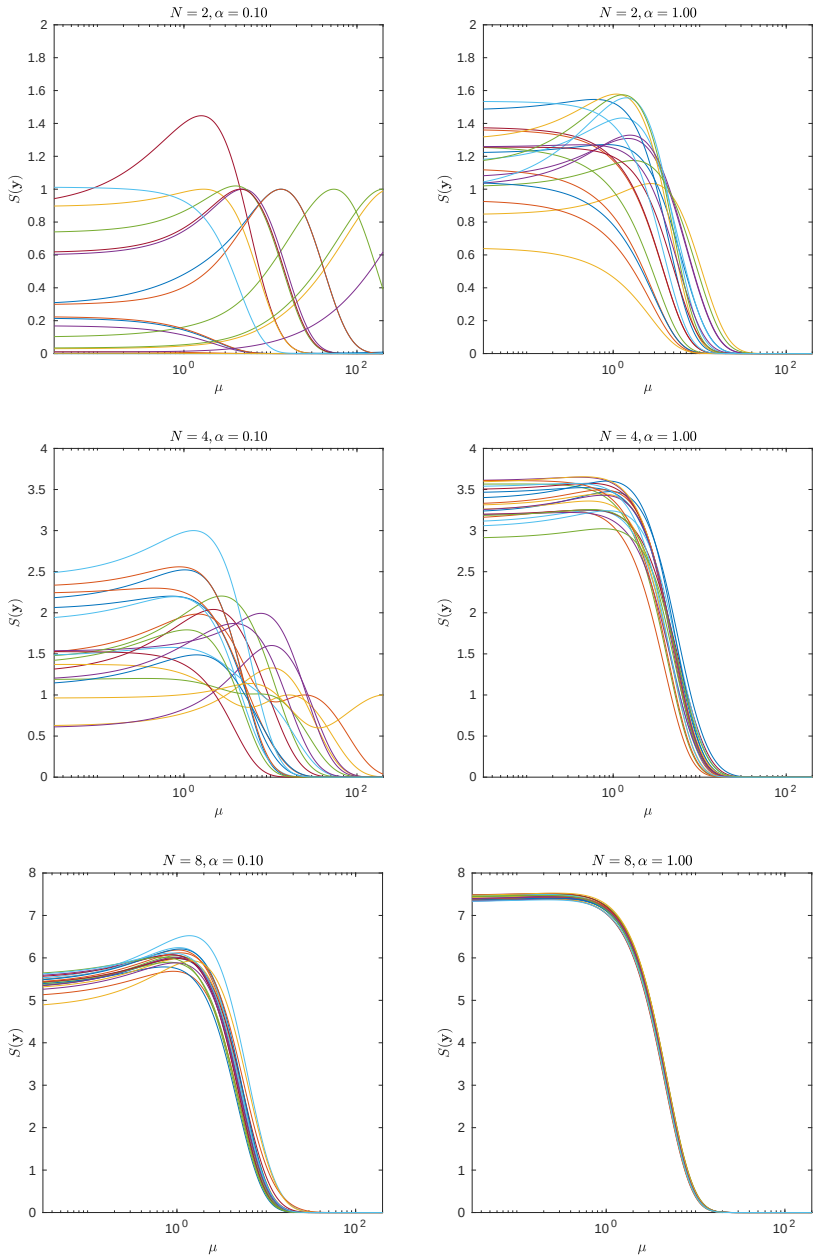

Figure 6.13: Shannon entropy $S(\mathbf{y})$ of the Dirichlet simulation realizations in different dimensions N running as a function of μ, where individual random realizations are visualized with different colors.

6.7 Summary

We built explicitly the following new constructions:

- ◇ A novel algebraic formulation for N-point correlations and complex event 'topologies' over rapidity, transverse momentum and multiplicity designed as new observables of high energy soft QCD diffraction. The framework includes vector fiducial partial cross sections, fractal marginal distributions, transverse momentum or multiplicity dependent gap trajectories and multidimensional diffraction fit and projection algorithms.
- ◇ A novel description of combinatorial compound processes based on the incidence algebras and inversion of the corresponding superposition Poisson problem based on the Möbius inversion theorem. The invertibility with finite statistics was simulated.

As a related technical remark regarding the superposition problem, the current event by event pileup correction methods at the LHC are essentially non-linear filtering operations (such as the median filter) applied to the observables event by event. A combination of the statistical approaches developed here and the event by event methods of [179–182] could be possible.

7 On the Inversion of High Energy Proton

Inversion of the K-fold stochastic autoconvolution integral equation is an elementary nonlinear problem, yet there are no de facto methods to solve it with finite statistics. To fix this problem, we introduce a novel inverse algorithm based on a combination of minimization of relative entropy, the Fast Fourier Transform and a recursive version of Efron's bootstrap. This gives us power to obtain new perspectives on non-perturbative high energy QCD, such as probing the ab initio principles underlying the approximately negative binomial distributions of observed charged particle final state multiplicities, related to multiparton interactions, the fluctuating structure and profile of proton and diffraction. As a proof-of-concept, we apply the algorithm to ALICE proton-proton charged particle multiplicity measurements done at different center-of-mass energies and fiducial pseudorapidity intervals at the LHC, available on HEPData. A strong double peak structure emerges from the inversion, barely visible without it.

Chapter in: arXiv:1905.12585 [hep-ph]

7.1 Introduction

Differential distributions of final state particles in high energy collisions, such as multiplicity distributions, are notoriously difficult to model precisely from the first principles of QCD, due to the nature of confinement and non-perturbative infrared physics. The topic has been studied since the early days of the Hagedorn fireball model [183], Feynman scaling [184] and Kobe-Nielsen-Olesen/Polyakov self-similarity (fractal) scaling [185, 186]. Hadron production has been treated as resulting from a breakdown of classic strings [187], the Feynman-Field model [188] and Lund strings of PYTHIA [189, 190], cluster hadronization of HERWIG [191], the topological expansion of cylindrical pomeron particle production model or 'quark-gluon strings' [192], to name some well known concepts. From the perturbative side, the local QCD parton-hadron duality [193] is perhaps the most appealing, with much support already from LEP data. Basically, it states that the hadronization happens at a low virtuality scale independent of the hard scattering scale. However, then the case of pure soft hadron-hadron scatterings without any hard scale involved is very interesting for collectivity or global color coherence reasons. These concepts are in one way or another implemented algorithmically in the Monte Carlo event generators, with a varying number of free parameters to be fixed by data, with goals of factorizing universal, the initial state and center-of-mass energy (in)-dependent properties.

In proton-nucleus (pA) and nucleus-nucleus (AA) physics a whole new class of phenomenological topics appear, from the space-time evolution of relativistic hydrodynamics and medium expansion to quark-gluon plasma signatures and the separation of different stages and collective phases of the process. A very interesting topic is the question which of these are due to heavy ion physics and which of them already happen in the elementary proton-proton interactions. To probe the ab initio physics behind many phenomenological models, and to factorize effects between heavy ions and elementary pp, solid mathematical approaches are crucial as analysis tools.

In this work, we introduce a novel inverse algorithm to invert a certain simple stochastic integral equation, the K-fold autoconvolution. By autoconvolution we mean convolving the distribution with itself and K-fold means repeating the operation recursively K-times. This is a nonlinear operation. In addition, we take the K to be a random number. The 'direct problem' and closely related problems are well studied in probability theory and implicitly ubiquitous in high energy physics, given the much studied pQCD logarithmic evolution integro-differential resummation schemes such as DGLAP in $\ln(Q^2)$ and BFKL in $\ln(1/x)$, which require as input the non-perturbative parton densities. The 'inverse problem' has been studied much less, considering the variety of methods for standard deconvolution and more generally,

methods to invert approximately linear and nonlinear integral equations with varying kernels. In experimental high energy physics, the unfolding of detector responses is more well known problem [194–197]. It corresponds to the usual deconvolution in the special case where the detector response matrix is of a convolution kernel type. One reason why inverse methods have not been studied extensively is, perhaps, the Monte Carlo event generators, which are extensively used as direct models. The demanding inverse problem, in that case, is the multidimensional parameter estimation or tuning of the event generator itself.

The problem of inverting the pileup effects in inclusive soft QCD multiplicity measurements by inverting a K-fold autoconvolution was treated in [52]. However, the subtractive iterative solution there was proposed without a statistical treatment, explicit regularization or uncertainty estimation and only for a small number K. We address these issues here by developing an all-order inverse. As far as we know, our approach which combines statistical modeling, Fast Fourier Transform (FFT) and Efron's bootstrap, is the first of its kind in high energy physics, but also including other fields. However, we mention that related problems have been studied also in a different context in laser optics [198], inverse problems and mathematics [199–203], and the problem is also closely related to the inverse Born series problem [204].

In Section 7.2 we introduce the mathematical direct problem and related formalism together with some simple examples and in Section 7.3 and 7.4 we go through the corresponding inverse problem, its discretization, regularization and the algorithmic solution. Section 7.5 is devoted to simulations and finally in Section 7.6 we do a proof-of-concept study of LHC-ALICE proton-proton multiplicity data. In Section 7.7 we discuss further applications and research directions.

7.2 Direct problem

Let the underlying continuous or discrete differential distribution be f_X, with normalization $\int f_X(x)\,dx = 1$ and the corresponding random variable be X. The distribution can be discrete, for example, in a case of charged particle multiplicity spectrum measurements. The K-fold autoconvolution means that the measurement g_Y is a sum of mutually independent identically distributed (i.i.d) random variables $Y = X_1 + X_2 + ... + X_K$. If we take $K \sim \text{Poisson}(\mu)$ and all $X_i \sim f_X$, this is known as a *compound Poisson* distribution. The convolution arises here naturally, because the sum of random variables is equivalent to a convolution between the probability distributions of the corresponding random variables.

The autoconvolution operation is defined with

$$[f_X \circledast f_X](y) \equiv \int_{-\infty}^{\infty} f_X(y-x) f_X(x)\,dx. \tag{7.1}$$

If we take the support of f_X on $[0, \infty)$, because physical observables are usually non-negative, then the autoconvolution integral range lower bound is zero. As a remark, in a different setup, a translation dependent Green's function $G(x, y)$ would be used instead of $f(x-y)$. That would give a Fredholm's integral equation instead of a convolution integral, leading us to the 'inverse Schrödinger equation' type of problems, as they are often called in the field of inverse problems.

For now, let us assume that the measurement follows a compound Poisson. However, in our algorithmic formulation we allow also any discrete distribution for K, not just Poisson distribution. The distribution of random variable Y is now given, by construction, as an autoconvolution series weighted with Poisson probabilities

$$\begin{aligned} g_Y(y) &= P_1 f_X(y) + P_2[f_X \circledast f_X](y) + P_3[[f_X \circledast f_X] \circledast f_X](y) + \dots \\ &= \frac{1}{1-e^{-\mu}} \sum_{n=1}^{\infty} \frac{\mu^n}{n!} e^{-\mu} f_X^{\circledast^n}(y), \end{aligned} \tag{7.2}$$

where the *convolution power* $\circledast^n$ is defined recursively as $f^{\circledast^n} = f^{\circledast^{(n-1)}} \circledast f$ and $f^{\circledast 1} = f$. We have removed the unobservable case $n = 0$ which gives $Y = 0$ and renormalized the remaining Poisson probabilities $P_n, n = 1, 2, 3, \dots$ to sum to one. The zero suppressed mean of the variable K is

$$\mathbb{E}[K > 0] = \mu/(1 - \exp(-\mu)) > 1, \tag{7.3}$$

where as the mean and variance (second central moment) of the compound distribution are given by useful formulas

$$\mathbb{E}[Y] = \mathbb{E}[K]\mathbb{E}[X] \tag{7.4}$$

$$\mathrm{Var}[Y] = \mathbb{E}[K]\mathrm{Var}[X] + \mathrm{Var}[K](\mathbb{E}[X])^2. \tag{7.5}$$

See [205] for all higher order moments and central moments of generic compound distributions.

In the Fourier spectral domain, the characteristic function (CHF) φ_X is defined as

$$\varphi_X(t) = \mathbb{E}[e^{itX}] = \int_{\mathbb{R}} e^{itX}\, dF_X(x) = \int_{\mathbb{R}} e^{itX} f_X(x)\, dx = \int_0^1 e^{itQ_X(p)} dp, \tag{7.6}$$

where $F_X(x)$ and $Q_X(p)$ are the cumulative and the inverse cumulative distribution functions of the random variable $X \in \mathbb{R}$. A probability generating function (PGF) G_X corresponding to a discrete probability mass function $p \sim X \in \mathbb{N}$ is a Laurent series around zero defined as

$$G_X(z) = \mathbb{E}[z^X] = \sum_{n=-\infty}^{\infty} p(n) z^n. \tag{7.7}$$

For the rest of the work we assume that the characteristic and generating functions are defined in the complex plane within their domain of convergence of the corresponding integral and sum, respectively, and leave the extensive algorithmic treatment of possible singularities for future work.

Using these tools, the well-known characteristic function for the compound Poisson is obtained with

$$\begin{aligned}
\varphi_Y(t) = \mathbb{E}[e^{itY}] &= \mathbb{E}_K\left[\left(\mathbb{E}[e^{itX}]\right)^K\right] = \mathbb{E}_K\left[\left(\varphi_X(t)\right)^K\right] \\
&= \sum_{n=0}^{\infty} \varphi_X(t)^n P(K=n) \\
&= \sum_{n=0}^{\infty} \varphi_X(t)^n \frac{\mu^n}{n!} e^{-\mu} \\
&= e^{\mu(\varphi_X(t)-1)}
\end{aligned} \tag{7.8}$$

and solving similarly for the case $K > 0$ $(n = 1, 2, ...)$ gives

$$\varphi_g(t) \equiv \varphi_{Y|K>0}(t) = \frac{e^{-\mu}\left(e^{\mu\varphi_f(t)} - 1\right)}{1 - e^{-\mu}} = \frac{1}{e^{\mu} - 1}(e^{\mu\varphi_f(t)} - 1), \tag{7.9}$$

where we denoted the characteristic function of $f_X(t)$ with $\varphi_f(t)$. A direct attempt to invert this would be to take the inverse Fourier transform, but we would encounter the problem of multivalued complex logarithm. Defining the complex logarithm in a suitable way was proposed together with a kernel estimator, for 'decompounding' in [202]. Here we take a different approach, where we *avoid* altogether using complex logarithms or other multivalued complex operations such as complex roots.

We will obtain a practical formal notation for the algorithm by representing the problem with an operator $\mathcal{F} : \Omega_X \to \Omega_Y$, where Ω_X and Ω_Y are the original and the smeared domain

$$\mathcal{F}(f) + \delta g = g. \tag{7.10}$$

Our main goal is to invert this nonlinear mapping taking also into account the statistical fluctuations δg. The nonlinearity comes directly from the fact that the autoconvolution operator $\mathcal{F}$ is constructed from f.

As a side remark, the existence of K-fold convolution is guaranteed by the *infinitely divisibility* of a probability distribution, which is the Lévy-Khintchine theorem. That is, for a given distribution f, there exists for any n, the corresponding n-th convolution power. All members of the stable family - for example Gaussian, Poisson, Negative binomial, Gamma, Cauchy (non-relativistic Breit-Wigner), Lévy and Landau - are infinitely divisible. They are also closed under convolution.

Examples

An interesting phenomenological fact, already observed in the CERN proton-antiproton UA5 data at $\sqrt{s} = 540$ GeV [206, 207], is that the charged particle multiplicity distributions tend to follow approximately a negative binomial distribution (NBD) or two of them, as the classic two component models may suggest. The same observation is still approximately valid at the LHC [208]. Mathematically speaking the NBD follows from a compound Poisson distribution of number of K *logarithmic series* $f(n;p) = p^n/(-n\ln(1-p)), n \in \mathbb{N}_+$ i.i.d variables each with shape parameter $p \in (0,1)$ while K being Poisson distributed with parameter $\mu \in \mathbb{R}_+$. Thus we obtain a negative binomial distribution on $\mathbb{N}_+$ as the compound distribution with parameters $1-p \in (0,1)$ and $-\mu/\ln(1-p) \in \mathbb{R}_+$. This mathematical picture may be useful to keep in mind while thinking the possible dynamical explanations.

Monte Carlo models reproduce NBD like distributions by multipomeron cuts or multiparton interaction modeling – the low multiplicity shape of the distribution is usually *very sensitive* to the non-perturbative proton profile or eikonal density driving the number K with impact parameter dependent distributions, and also to the diffraction contribution. Kinematically, a system with the cms energy W will

span a longitudinal rapidity range $Y \sim \ln(W^2/W_0^2)$, where W_0 is at the order of the proton mass. In Lund model like scenarios, also the average number of particles scales like $\langle N \rangle \sim \ln(W^2/W_0^2)$. A multistring system total multiplicity consisting of K-strings will behave as $\langle N \rangle \sim \sum_{i=0}^{K-1} \ln(s_{i,i+1}/W_0^2)$ with $s_{i,i+1} = (k_i + k_{i+1})^2$ being sub-Mandelstam invariants constructed from the parton 4-momenta $\{k_i\}$ spanning the strings and the variance $\mathrm{Var}[N]$ behaves proportionally to $\langle N \rangle$ [209], which is a Poisson like behavior. So if the sub-W^2 distributions behave typically like falling power laws and the number of K-simultaneous strings are also modeled as Poisson like with impact parameter dependence or not, one will generate negative binomial like distributions. Let us emphasize at this point, the soft transverse momentum dependence with multiplicity is not understood from first principles.

Now let us think instead of the experimentally visible superposition of 'pileup' proton-proton interactions at the LHC for a given Poisson μ. At the LHC Poisson μ is measured for low values of $\mu \sim 0.05$ (ALICE) using the minimum bias trigger occupancy[1] $\langle R \rangle = 1 - P_0(\mu) = 1 - e^{-\mu} \Leftrightarrow \mu = -\ln(1 - \langle R \rangle)$, where $R \in [0, 1]$. For large values of $\mu \sim 50$ when the occupancy is completely saturated (ATLAS, CMS), the scaling of the track multiplicity or the number of primary vertices is used to determine μ. A combination of both can be useful for intermediate values of $\mu \sim 5$ (LHCb), for example. The pileup is illustrated in Figure 7.1. The convolution of a negative binomial with a negative binomial is yet again a negative binomial – this is the closure under convolution. It is worth pointing out that the estimation of μ based solely on the measured distribution $g(x)$ itself cannot be done in general. This is because if the distribution is from the family of distributions which are closed under convolutions, then we have no capability of identifying if the measured distribution is pileup free, or, one with a larger scale and position parameter than the expected one, or the autoconvoluted version of one with smaller scale and position parameter. Thus, μ must be a parameter of the inversion and must be inferred from other measurements or theoretical constructions.

Clearly, our discussion is now very close to the KNO / Polyakov scaling [186], which states that there is one elegant scaling function ψ giving the probability of multiplicity n as a function of the scaling variable $n/\langle n(s) \rangle$

$$P(n; s) = \frac{1}{\langle n(s) \rangle} \psi \left(\frac{n}{\langle n(s) \rangle} \right), \tag{7.11}$$

where s is the center of mass energy squared Mandelstam variable, a Lorentz scalar. However, this scaling is known to be violated at some level since the ISR times, and

[1] $\langle R \rangle_{t,N_B} = \frac{\frac{1}{\Delta t} \int_{\Delta t} f_R(t)\, dt}{N_B f_O}$, where N_B is the number of colliding bunch pairs, f_O the LHC revolution frequency (Hz) and $f_R(t)$ the instantaneous trigger rate (Hz).

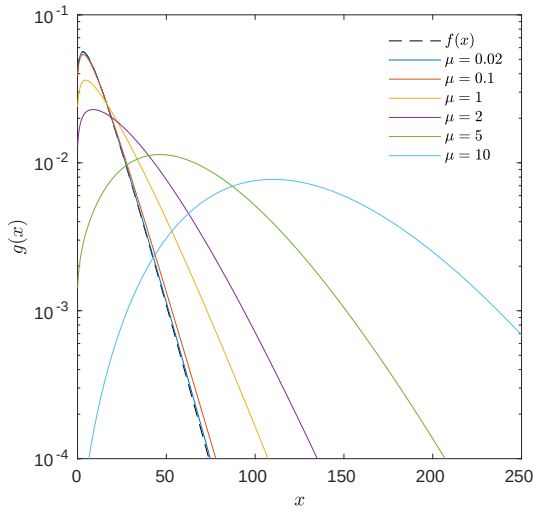

Figure 7.1: K-fold compound Poisson autoconvolution of the negative binomial distribution with varying μ.

at the LHC it holds merely in limited regions of the full phase space, in a narrow central rapidity window. In general, when inspecting scaling or its violation one needs to take into account the collision initial state, e.g. pp versus e^+e^-, the fiducial phase space of the measurement and contribution of diffraction. At the model level, multiparton or multipomeron exchanges running as a function of energy are usually creating significant deviations from the KNO scaling. Additional effects arise from soft versus hard scale interactions. See the books [210, 211] for a discussion on a variety of perturbative QCD and other scenarios exhibiting scaling.

By the central limit theorem, the K-fold convolution will map almost[2] every distribution eventually into a Gaussian distribution when $\mathbb{E}[K] \to \infty$. This is clearly visible in Figure 7.1, where we used the direct mapping of Equation 7.2 with different Poisson μ-values to convolve the negative binomial distribution

$$P_{\mathrm{NBD}}(n\,;\,\langle n\rangle, k) = \frac{\Gamma(k+n)}{\Gamma(k)\Gamma(n+1)}\left[\frac{\langle n\rangle}{k+\langle n\rangle}\right]^n\left[\frac{k}{k+\langle n\rangle}\right]^k \tag{7.12}$$

with a fit to the LHC charged particle multiplicity data in proton-proton at $\sqrt{s} = 7$ TeV for pseudorapidity range $\eta \in [-0.9, 0.9]$ obtained from [208]. The fit parameters

[2]For example Cauchy / non-relativistic Breit-Wigner type distributions do not turn into Gaussian, also moments for these distributions are undefined.

central values are $\langle n \rangle = 12.5$ and $k = 1.4$, where $\langle n \rangle$ is the average multiplicity of NBD and k is related to the variance $D^2 = \langle n^2 \rangle - \langle n \rangle$ by $D^2/\langle n \rangle^2 = 1/\langle n \rangle + 1/k$.

In an explicit experimental realization, to take into account the detector response and its effects on the resolution and efficiency, an unfolding algorithm with uncertainty estimation and other corrections should be processed first – if necessary. The output of this can be then propagated to the algorithm described here. For the rest of the work, we assume this to be the case.

7.3 Inverse problem

Solving the K-fold autodeconvolution means here a statistical inverse of Equation 7.10. Unfortunately, the measurement noise and modeling errors will usually always forbid us executing a direct inversion in practice, even if we would have a closed-form expression to do so. This ill-posedness requires usually either implicit or explicit regularization which turns the problem into a more well-posed one.

We calculate the convolution by pointwise multiplication in the Fourier domain using the convolution theorem $f \circledast g = \mathfrak{F}[f]\mathfrak{F}[g]$, which works also the other way around $\mathfrak{F}[f] \circledast \mathfrak{F}[g] = fg$, which is also often called ambiguously convolution or folding of distributions of f and g. Now, a fixed $K \in \mathbb{N}$ autoconvolution is given by

$$\mathfrak{F}[f^{\circledast K}] = (\mathfrak{F}[f])^K = \mathfrak{F}[g], \tag{7.13}$$

where the power is applied as $z^K = (|z|e^{i(\varphi+2\pi n)})^K = |z|^K e^{iK\varphi} e^{2\pi i n K}, n \in \mathbb{Z}$. Thus, in the spectral domain the stochastic autoconvolution results in a random scaling of amplitudes and a rotation of phases of the characteristic function. Now when K is not an integer this results in a multi-valued operation, because $e^{2\pi i n K}$ does not map a full rotation around the complex plane for different values of n. As a reminder, if we shift the probability density $f(x-x_0)$, this results in a phase shift in the frequency domain by $\exp(-i\omega x_0)\mathfrak{F}[f]$.

A formal 'naive' inverse is obtained by taking the complex K-th root in the Fourier domain, and then proceeding with an inverse Fourier transform

$$\left(\mathfrak{F}[f]^K\right)^{1/K} = (\mathfrak{F}[g])^{1/K} \tag{7.14}$$

$$f = \mathfrak{F}^{-1}\left[(\mathfrak{F}[g])^{1/K}\right], \tag{7.15}$$

analogously to the standard naive Fourier deconvolution. However, complex root is a multivalued operation $|z|^{1/K} e^{i(\mathrm{Arg}(z)+2\pi n)/K}$ with $n = 0, 1, \ldots, K-1$ roots with suitable branch cuts to be specified algorithmically, in principle. Now we see that the only strictly well posed case is the case when the characteristic function is real and non-negative $\leftrightarrow$ a mirror symmetric (even) probability density around zero by the properties of the Fourier transform, a case which is clearly too restrictive to be of interest here. Also, this 'solution' does not take into account that K is a random variable, neither it takes into account the finite statistics of g and mismatches between the measurement and modeling causing instabilities to be regulated. For some related discussion, see for example [212].

Discretization

To solve the finite sample version of the problem, we need a (discrete) representation of the problem. For simplicity, computational efficiency and due to the high energy physics analysis convention – as a practical format of the measurement, we use histograms as the density estimators of f and g, even if simple kernel (Parzen) estimators and even more advanced density estimation techniques could be used in principle. Also, we consider only a 1D-version of the problem. However, extensions to higher dimensions are formally straightforward.

The histograms are defined in terms of D, KD bins with corresponding fixed bin width $\Delta x \equiv \Delta y$. In the context of algorithms histograms are represented with finite non-negative column vectors, which we denote with bold symbols

$$\mathbf{f}[n] = |\{f_i \in [x_n, x_n + \Delta x)\}|\,, \quad n \in \mathbb{Z}_{\Omega_X} := \{0, 1, \ldots, D-1\} \tag{7.16}$$

and similarly for a K-fold autoconvoluted

$$\mathbf{g}[n] = |\{g_i \in [y_n, y_n + \Delta y)\}|\,, \quad n \in \mathbb{Z}_{\Omega_Y} := \{0, 1, \ldots, K(D-1)\}, \tag{7.17}$$

where $f_i, g_i \in \mathbb{R}$ denote sample elements and x_n, y_n denote the bin lower edges. We extended the domain of autoconvoluted by K-times and because K is a random variable, in practice we take some large enough integer which avoids any truncation problems in the upper tail. Only values of $\{g_i\}$ are directly observable and measured. Now the discrete autoconvolution is defined as

$$[\mathbf{f} \circledast \mathbf{f}][n] = \sum_{m=0}^{2(D-1)} \mathbf{f}[m]\mathbf{f}[n-m], \tag{7.18}$$

and the K-fold version follows recursively. In the algorithmic implementations, when necessary to extend the domain of vectors due to convolution, we do it explicitly by zero padding.

In principle, the binning can be also non-uniform, such as logarithmic. However, that must be taken explicitly into account in the evaluation of the discrete convolution, which is defined using uniform sampling. One classic solution could be interpolation and resampling schemes with B-splines, for example. Here, however, we use only fixed binning. The bin width Δx needs to be chosen such that the spectrum resolution criteria and event count statistics are taken into account. This in order not the create a 'noisy spectrum' with low bin counts, or on the other hand, induce a loss of resolution. Thus, it has also a role as an implicit regulator.

Relative entropy minimization

The measured histogram $\mathbf{g}$ is assumed to be bin-by-bin Poisson distributed, a usual assumption, given also the Poisson superposition property $\sum_i \text{Poisson}(\mu_i) \sim \text{Poisson}\left(\sum_i \mu_i\right)$. The Poisson likelihood with a known constant background histogram $\mathbf{b}$ is then written as a product over the histogram bins

$$p(\mathbf{g}|\mathbf{f}) = \prod_i \frac{[\mathcal{F}\mathbf{f}+\mathbf{b}]_i^{g_i} e^{-[\mathcal{F}\mathbf{f}+\mathbf{b}]_i}}{g_i!} \tag{7.19}$$

and its negative log-likelihood is now our fit quality term

$$J(\mathbf{f}|\mathbf{g}) = -\log p(\mathbf{g}|\mathbf{f}) = \sum_i -g_i \log([\mathcal{F}\mathbf{f}+\mathbf{b}]_i) + [\mathcal{F}\mathbf{f}]_i + b_i + \log(g_i!) \tag{7.20}$$

and by neglecting terms which do not affect the solution and rearranging gives

$$J(\mathbf{f}|\mathbf{g}) = \sum_i g_i \frac{g_i}{\log([\mathcal{F}\mathbf{f}+\mathbf{b}]_i)} + [\mathcal{F}\mathbf{f}]_i - g_i. \tag{7.21}$$

This often used formulation is the minimum entropy solution, and is equivalent to minimizing generalized Kullback-Leibler divergence (relative entropy) [213], known also as the Csiszár I-divergence [214]. The gradient of this functional is

$$\nabla J(\mathbf{f}) = \frac{\partial}{\partial \mathbf{f}} \left(-\mathbf{g}^T \log(\mathcal{F}\mathbf{f}+\mathbf{b}) + 1^T \mathcal{F}\mathbf{f}\right) = \mathcal{F}^T\mathbf{1} - \mathcal{F}^T \frac{\mathbf{g}}{\mathcal{F}\mathbf{f}+\mathbf{b}} = \mathcal{F}^T\left(\mathbf{1} - \frac{\mathbf{g}}{\mathcal{F}\mathbf{f}+\mathbf{b}}\right), \tag{7.22}$$

where divisions are understood component wise. If distributions are normalized to unity, then $\mathcal{F}^T\mathbf{1} \equiv \mathbf{1}$ holds. Here, we use event counts.

The gradient based iterative update or a fixed point iteration is now obtained from the steepest descent update $\mathbf{f}_{k+1} = \mathbf{f}_k - \gamma D_k \nabla J(\mathbf{f}_k)$, where D_k is a gradient scaling matrix. If we set $D_{\mathbf{f}_k} = \text{diag}(\mathbf{f}_k)$ and $\gamma = 1$ which is well known to correspond in this case to the expectation maximization (EM) [215] formulation of a frequentist maximum likelihood solution under the Poisson likelihood, then we obtain a Richardson-Lucy [216, 217] type multiplicative deconvolution formula well known in optics and astrophysics

$$\mathbf{u}_k = \left(\mathbf{f}_k \oslash (\mathcal{F}^T\mathbf{1})\right) \odot \left(\mathcal{F}^T\mathbf{g} \oslash (\mathcal{F}\mathbf{f}_k + \mathbf{b})\right), \tag{7.23}$$

where $\odot$ and $\oslash$ are Hadamard's vector component wise product and division, respectively. This was also re-invented by D'Agostini [197] in high energy physics

unfolding context. For the rest of the paper, we set the background $\mathbf{b} = 0$. The RL-type update conserves non-negativity of the solution and the total number of events. Another way to derive the RL type update is to set gradient Equation 7.22 to zero, move terms on both sides, multiply by $\mathbf{f}$ and do fixed point iteration.

Regularization

The problem of choosing the spectral bandwidth cut-off or regularization strength in the non-transformed domain are the most common but also the most difficult topics of inverse problems. Motivated by the fact that the distributions of observables for us are usually smooth and non-discontinuous, we use a traditional variational regularization with a Tikhonov ℓ_2-norm type regularity functional

$$J_R(f) = \lambda \int_\Omega \|\nabla f(x)\|^2 \, dx, \tag{7.24}$$

where $\Omega \subset \mathbb{R}$. The minimum of this is given by

$$\begin{aligned} \nabla J_R(f) &= \lambda \nabla \left(\int_\Omega \|\nabla f(x)\|^2 dx \right) \\ &= \lambda \nabla^* \nabla f(x) = -\lambda \mathrm{div}(\nabla f(x)) = -\lambda \nabla^2 f(x) = 0, \end{aligned} \tag{7.25}$$

where we applied to the integral von Neumann boundary conditions and Gauss divergence-theorem, the standard vector analysis identities and ∇^2 is the usual Laplacian. That is, the gradient descent direction is given by the negative Laplacian.

Instead of using the abstract functional gradient, the regularization term is discretized using a finite difference matrix and the corresponding finite Laplacian

$$\nabla_M \equiv \begin{pmatrix} 1 & -1 & 0 & \cdots & 0 \\ 0 & 1 & -1 & & \vdots \\ \vdots & & & \ddots & 0 \\ 0 & \cdots & 0 & 1 & -1 \end{pmatrix}, \; \nabla_M^2 \equiv \nabla_M^T \nabla_M = \begin{pmatrix} -1 & 2 & -1 & \cdots & 0 \\ 0 & -1 & 2 & -1 & \vdots \\ \vdots & & & \ddots & 0 \\ 0 & \cdots & -1 & 2 & -1 \end{pmatrix}. \tag{7.26}$$

It is naturally also possible to modify the boundary conditions which affect these matrices, based on a priori assumed or known properties of the distribution $f(x)$.

Regularization parameter λ depends on the number of observed events count N, the number of histogram bins D (= problem discretization) and the shape of the distribution of interest. The average pileup μ and the functional smoothness of the distribution under inversion are very important factors affecting the nonlinear direct

operator and thus the necessary regularization strength. In the limit $N_E \to \infty$ and $\mu \to 0$, we can take asymptotically $\lambda \to 0$. There are several semi-heuristic 'data-driven' methods developed for selecting the regularization. The well-known principles are the L-curve, generalized cross validation (GCV), Morozov's discrepancy principle and various covariance and residual spectrum analysis methods [218]. Monte Carlo event generator driven or a toy model based analysis together with data driven approaches is also a reasonable choice in high energy physics. We use a simple variational equilibrium approach to choose the regularization parameter. This is is described in Section 7.5.

The full solution taking together fidelity + regularity usually means directly minimizing in the direction of combined gradient of the full cost

$$\min_{\mathbf{f}} \; J(\mathbf{f}|\mathbf{g}) + \lambda J_R(\mathbf{f}) \tag{7.27}$$

as implemented for example in [219] in the context of image deconvolution. However, this we observed to give very unstable results in this problem. Thus, we implemented the regularization step through forward-backward slitting type gradient update, similar in fashion to proximal optimization algorithms. It means that we update the result after the EM-update step as

$$\mathbf{f}_{k+1} = \mathcal{P}_+ \left(\mathbf{u}_k - \lambda \nabla^2_M \mathbf{u}_k \right), \tag{7.28}$$

where $\mathcal{P}_+(\cdot)$ is the positive solution projector operator $\mathcal{P}_+ : \mathbb{R}^n \to \mathbb{R}^n_+$ such that $\mathcal{P}_+(\mathbf{x}) = \mathbf{y}$ gives $y_i = x_i$ if $x_i > 0$, else, $y_i = 0$, $\forall i = 1, \ldots, n$. Separating the regularization as a separate step resulted in a much improved behavior. However, the approach chosen here is the best performing we found from the vast phase space of optimization algorithms and regularization approaches. One must remember that we are not dealing here with an inverse problem with a fixed kernel, thus this problem has its own peculiarities.

7.4 Algorithm

For every iteration, we construct the autoconvolution operator by taking the discrete Fourier transform using FFT as $F = \mathfrak{F}[\mathbf{f}]$ and then construct the empirical characteristic function of the projected measurement with a nonlinear complex map $G = \frac{1}{e^{\mu}-1}\left(e^{\mu F}-1\right)$ using Equation 7.9, where complex exponential is the single valued function $\exp(z) = \sum_{n\geq 0} z^n/n!$, $z \in \mathbb{C}$. The autoconvolution 'kernel' in the spectral domain is then obtained by taking a point wise division $H = G \oslash F$, inverse transforming $\mathbf{h} = \mathfrak{F}^{-1}[H]$ and finally representing it as a Toeplitz convolution matrix to obtain $\mathcal{F} = \mathcal{T}[\mathbf{h}]$. The operator $\mathcal{T}$ represents here a simple standard way to construct a Toeplitz matrix. We assume F to be non-zero for all of its components, which might be violated in pathological cases. These are intrinsically non-invertible without extra a priori information.

We remark here that this nonlinear map and division in the spectral domain has a different goal than in the usual Fourier domain linear deconvolution such as in Wiener filters [220], where the noise amplification is regularized in the spectral domain. In Wiener filtering, the division gives a solution to the problem non-recursively. Here, on the other hand, we use the spectral domain as a way to obtain the Toeplitz matrix representation in the probability domain extremely efficiently via FFT. The full solution is recursive and regularized in the probability domain.

Discretization of the operator $\mathcal{F}$ in the case of arbitrary (non-Poisson) compound distribution is done with a finite number of convolution Toeplitz matrices $F^{(n)}, n = 1, \ldots, \tilde{K}$ with $F^{(1)} = I$ up to a numerically suitable finite order $\tilde{K}$. The maximum order is basically limited only by the available computing resources. These Toeplitz matrices are then added together with weights P_n. That is, we evaluate the discretized and truncated version of Equation 7.2. This numerical finite order implementation was cross checked against the exact all order Fourier domain method in the Poisson case, as described above, and they agreed within floating point accuracy.

Now summarizing: the multiplicative algorithm describe above corresponds to a non-negative solution with Poisson bin fluctuations likelihood with the generalized Kullback-Leibler divergence being the fit criteria. On the other hand, subtractive algorithms are usually encountered with iterative solutions to unconstrained optimization under Gaussian noise and ℓ_2-minimum norm minimizing the least squares cost $J_{\mathrm{LS}}(\mathbf{f}) = \frac{1}{2}\|\mathcal{F}\mathbf{f} - \mathbf{g}\|^2$ with an iterative solution

$$\mathbf{f}_{k+1} = \mathbf{f}_k - \gamma \nabla J_{\mathrm{LS}} = \mathbf{f}_k - \gamma \mathcal{F}^T\left(\mathcal{F}\mathbf{f}_k - \mathbf{g}\right) \tag{7.29}$$

which is known as the Landweber iteration scheme [221]. Non-negativity of the solution should be enforced simply with the projector $\mathcal{P}_+(\cdot)$, because it is not a

Algorithm 1 Kisu: K-fold Inverse of Stochastic Autoconvolution

INPUT: Smeared histogram $\mathbf{g}$ with N events, regularization strength $\lambda \in \mathbb{R}_+$, Poisson mean $\mu \in \mathbb{R}_+$ **OR** discrete distribution P_n with $\sum_{n=1}^{\tilde{K}} P_n = 1$, number of iterations $R \in \mathbb{N}_+$, background histogram $\mathbf{b}$ or otherwise $\mathbf{b} = 0$.

If μ given, use option **§I** below, **else**, use input P_n and option **§II** below
Initialize $\mathbf{f}_1 \leftarrow \mathcal{Z}_{\tilde{K}}(\mathbf{g} - \mathbf{b})$ with $\tilde{K}$-fold zero padding $\mathcal{Z}_K(\cdot)$

for all $k = 1, \ldots, R$ **do**

Step 1. Construction of the direct operator:

§I: Poisson case; 'all-order' map:
$F \leftarrow \mathfrak{F}[\mathbf{f}_k]$, $G \leftarrow \frac{1}{e^{\mu}-1}\left(e^{\mu F} - 1\right)$, $H \leftarrow G \oslash F$, $\mathbf{h} \leftarrow \mathfrak{F}^{-1}[H]$, $\mathcal{F}_k \leftarrow \mathcal{T}[\mathbf{h}]$

§II: Generic P_n case; evaluated 'order-by-order':
for all $n = 2, \ldots, \tilde{K}$ **do**
$F_k^{(n)} \leftarrow \mathcal{T}[\mathbf{f}_k^{\circledast^{n-1}}]$ # Construct convolution Toeplitz matrix
$\mathbf{f}_k^{\circledast^n} \leftarrow \mathbf{f}_k^{\circledast^{n-1}} \circledast \mathbf{f}_k$ # Calculate convolution power
end for
$\mathcal{F}_k \leftarrow \sum_{n=1}^{\tilde{K}} P_n F_k^{(n)}$, where $F_k^{(1)} = I$ # Weighted sum of convolution matrices

Step 2. EM-step + regularization/smoothing with positivity constraint:
$\mathbf{u}_k \leftarrow \left(\mathbf{f}_k \oslash (\mathcal{F}^T\mathbf{1})\right) \odot \left(\mathcal{F}^T\mathbf{g} \oslash (\mathcal{F}\mathbf{f}_k + \mathbf{b})\right)$
$\mathbf{f}_{k+1} \leftarrow \mathcal{P}_+\left(\mathbf{u}_k - \lambda\nabla_M^2\mathbf{u}_k\right)$
end for

OUTPUT: Deconvolution estimate $\hat{\mathbf{f}} \leftarrow \mathcal{Z}_{\tilde{K}}^{-1}(\mathbf{f}_R)$ with the same support as $\mathbf{g}$.

built-in constraint of the standard least squares. The γ is a relaxation parameter that controls the convergence and can be optimized by several different line search methods or by fixing it to a constant ~ 1, which gives (much) slower convergence. This Gaussian version of the algorithm could be utilized with large event samples. With low event count histograms, the multiplicative solution was observed to be superior in terms of the stability of the solutions.

Uncertainty estimation

Point estimates are obtained with Algorithm 1, which we call Kisu. The algorithm first solves the direct problem and then one EM-step plus regularization of the inverse, and recursively alternate between these two until convergence. The regularization is done explicitly, because it is well known that the EM-type maximum likelihood solution alone will amplify the high frequencies ($\sim$ Poisson noise) when the number of iterations $k \to \infty$. That is, iteration first fits the low frequency Fourier components and later starts to fit the high frequency noise to the solution. A semi-explicit regularization strategy would correspond to an early iteration stop. However, with an explicit regularization scheme as here, the iteration can be allowed to continue till convergence within some numerical threshold.

The iterative bias estimation and uncertainty estimation are done using numerical Monte Carlo bootstrap with Algorithms 2 and 3. We call these the daughter and mother bootstrap, respectively. The algorithm is started by calling the mother bootstrap, which calls the daughter bootstrap, which calls Kisu. The bootstrap is selected here due to the nonlinear and recursive nature of the problem which makes the usual linearized Taylor expansion error propagation and analytical (Gaussian) approximations unreliable or semi-impossible. The bootstrap was introduced in the seminal paper by Efron in 1979 [222] and the recursive bias correction was first discussed in 1986 [223]. In high energy physics unfolding context, bootstrap iterated bias correction and related confidence intervals were utilized only quite recently in [195] and we follow a similar strategy. The underlying inverse problem is nonlinear here, in contrast. The bootstrap is a relatively easy method to implement and variants of it have already long been used in high energy physics dubbed often under the large umbrella of 'toy Monte Carlo'. A practical problem is the high computational cost, fortunately, bootstrap sampling is trivially parallelizable. However Algorithm 1 and iterative bias correction loop of Algorithm 2 are recursive, and thus cannot be made fully parallel.

The bias of a parameter estimator is by definition $\mathrm{Bias}[\hat{\theta}] = \mathbb{E}[\hat{\theta}] - \theta$. An iterative bootstrap estimate of the bias replaces the unknown expectation with a

Algorithm 2 'Daughter bootstrap' – Iterative Bias Correction

INPUT: Smeared histogram $\mathbf{g}$, number of bias correction iterations $M \in \mathbb{N}_+$, bootstrap sample size per iteration $B \in \mathbb{N}_+$ and Algorithm **A1** parameters.

Run **A1** using $\mathbf{g}$ as input, obtain $\hat{\mathbf{f}}_0$
Set $\hat{\mathbf{f}}_\star \leftarrow \hat{\mathbf{f}}_0$, $\hat{\mathbf{g}}_\star \leftarrow \mathbf{g}$
for all $i = 1, \ldots, M$ **do**
  Set $S_i \leftarrow \emptyset$
  for all $j = 1, \ldots, B$ **do**
    Draw $\mathbf{g}_j^* \sim \hat{\mathbf{g}}_\star$ with replacement using toy Monte Carlo
    Run **A1** using $\mathbf{g}_j^*$ as input, obtain $\mathbf{f}_j^*$, update $S_i \leftarrow S_i \cup \{\mathbf{f}_j^*\}$
  end for
  Step 1. Iterated bias estimate using bootstrap sample S_i mean:
  $\hat{\text{Bias}}[\hat{\mathbf{f}}] \leftarrow \text{Mean}[S_i] - \hat{\mathbf{f}}_\star$
  Step 2. Point estimate with bias correction, and non-negativity re-enforced:
  $\hat{\mathbf{f}}_\star \leftarrow \mathcal{P}_+(\hat{\mathbf{f}}_0 - \hat{\text{Bias}}[\hat{\mathbf{f}}])$
  Step 3. Using direct operator estimation **Step 1.** of **A1**, generate:
  $\hat{\mathbf{g}}_\star \leftarrow \mathcal{F}_\star(\hat{\mathbf{f}}_\star)$
end for

OUTPUT: Bias corrected estimate $\hat{\mathbf{f}}_\star$, first estimate $\hat{\mathbf{f}}_0$ and the sample S_B for the standard bootstrap confidence interval evaluation.

Algorithm 3 'Mother bootstrap'

INPUT: Smeared histogram $\mathbf{g}$, bootstrap sample size $Q \in \mathbb{N}_+$, Algorithm **A2** and **A1** parameters.

Set $S_\star \leftarrow \emptyset, S_0 \leftarrow \emptyset$
for all $q = 1, \ldots, Q$ **do**
  Draw $\mathbf{g}_q^* \sim \mathbf{g}$ with replacement using toy Monte Carlo
  Run **A1** using $\mathbf{g}_q^*$ as input, obtain $\hat{\mathbf{f}}_{\star q}^*$ and $\hat{\mathbf{f}}_{0q}^*$, update $S_\star \leftarrow S_\star \cup \{\hat{\mathbf{f}}_{\star q}^*\}, S_0 \leftarrow S_0 \cup \{\hat{\mathbf{f}}_{0q}^*\}$
end for

OUTPUT: Samples $S_\star$ and S_0 for calculating the confidence intervals, median etc., for both bias corrected ($\star$) and uncorrected (0) estimates.

bootstrap sample mean and the true parameter value with the current estimate of the parameter. This is described in algorithm 2. However, the variance of the estimator $\mathrm{var}[\hat{\theta}] = \mathbb{E}[(\hat{\theta}) - E[\hat{\theta}])^2]$ can grow as a result of the bias correction. That is, thinking in terms of classic but non-robust mean squared error $\mathrm{MSE}[\hat{\theta}] = \mathbb{E}[(\hat{\theta} - \theta)^2] = \mathrm{var}(\hat{\theta}) + (\mathrm{Bias}[\hat{\theta}])^2$, it is clear that a good estimator is a combination of both qualities. The classic goal has usually been to find out the minimum variance unbiased estimator (MVUE), which clearly minimizes MSE among the unbiased estimators. For more information about the iterative bootstrap, we refer the reader to the book by Hall [224].

We calculate the uncertainty estimates by ordering the sample obtained from the mother algorithm and calculate the vector component (pointwise) $1 - 2\alpha$ *percentile intervals* at level α as $[\hat{\mathbf{f}}^*_{\star/0,\alpha}, \hat{\mathbf{f}}^*_{\star/0,1-\alpha}]$, where $\star/0$ means we consider both bias corrected ($\star$) and uncorrected (0) estimates and their intervals. As another option, the so-called *basic bootstrap intervals* would be obtained with $[2\hat{\mathbf{f}} - \hat{\mathbf{f}}^*_{\star/0,1-\alpha}, 2\hat{\mathbf{f}} - \hat{\mathbf{f}}^*_{\star/0,1-\alpha}]$, where $\hat{\mathbf{f}}$ is the estimate obtained directly running Kisu. Basic bootstrap intervals have clearly a possibility of flipping the intervals upside down, which we observed also in the simulations. One could also go further and estimate the spectrum global uncertainty coverage, not just point by point local one. For more information, see [225].

7.5 Simulations

The first scenario to study is the simplest fixed K autoconvolution inverse of the negative exponential distribution without any uncertainty estimation. We illustrate this in Figure 7.2 and compare with the naive Fourier domain inverse with spectral cut-off regularization, that is, we set high frequencies above the cut-off Λ_ω to zero, with $\omega \in [0, 1]$. The spectral cut-off is tuned to give the best performance for this scenario. We observe nearly perfect reconstruction with KISU and very large oscillations with the naive Fourier domain algorithm. These oscillations seemed to appear immediately after a small amount of counting fluctuations in the observed spectrum g, which we simply added here as Gaussian noise. For a typical iteration trajectories, see Figure 7.3. We see that the variational regularity cost is kept as almost constant, and the error to the ground truth and the re-projection error behave in the same way, which is always called for behavior.

We tried to two different 'data-driven' ways of choosing the regularization parameter, see Figure 7.4. First, the regularization parameter λ was selected as the equilibrium point $\min(\alpha + \beta)$, where the re-projection error is $\alpha = \chi^2_{\hat{g}} = ||(\mathbf{g} - \hat{\mathbf{g}})/\sqrt{\mathbf{g}}||^2_{\ell_2}$ and the regularity cost is $\beta = \chi^2_{\nabla^2 \hat{f}} = ||\nabla^2 \hat{\mathbf{f}}/\sqrt{\hat{\mathbf{f}}}||^2_{\ell_2}$, with operations taken element wise. Re-projection is simply defined as $\hat{g} = \hat{\mathcal{F}}(\hat{f})$, available after the inversion. The

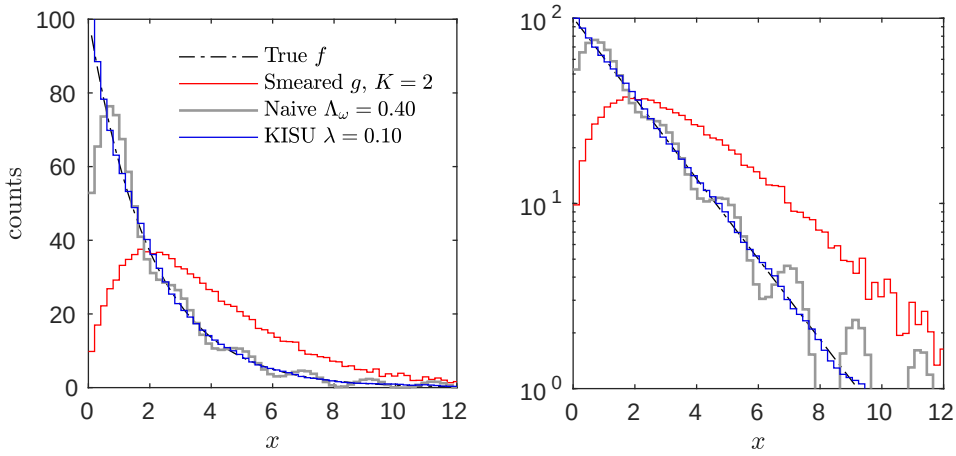

Figure 7.2: Simplified fixed K autoconvolution inverse using the naive Fourier domain inverse with spectral cut-off and KISU based inverse. The true distribution is $f(x) = 1/\alpha \exp(-x/\alpha)$ with $\alpha = 2$.

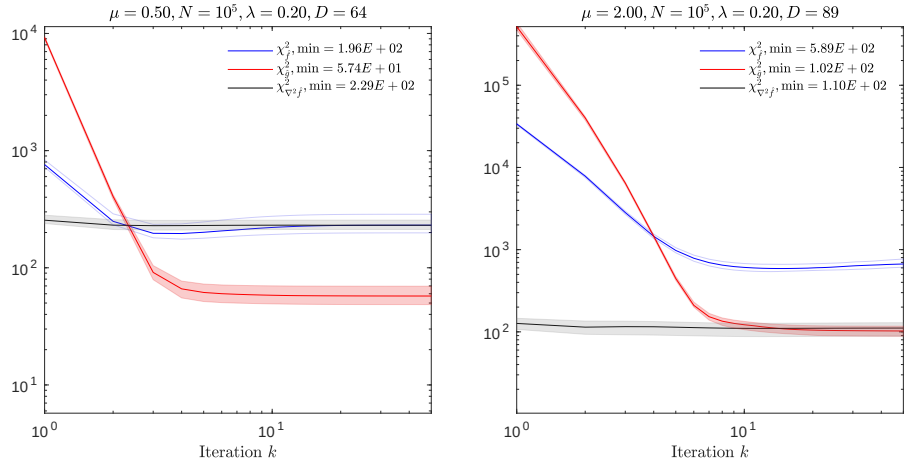

Figure 7.3: Typical Kisu algorithm iteration trajectories. Red lines denote the re-projection error, blue lines are the error with respect to the true distribution (available only in simulation) and black lines denote the variational regularity (smoothness).

values for different λ values are obtained via brute force loop. This was repeated for each simulation scenario separately. The second criteria was to use the Morozov like discrepancy principle point where the re-projection error α is approximately the same as the number of histogram bins D. For the rest of the simulations, we use the equilibrium solution. When $\lambda \to 1$, the solution becomes over smooth which results in a loss of high frequency details and when $\lambda \to 0$, we start to fit the noise in high frequencies. In general, slightly larger λ values were always obtained as the equilibrium solution compared to the discrepancy principle.

In more demanding simulations we have a scenario that corresponds approximately to inverting minimum bias pileup at the LHC. We chose fully analytical distributions as the ground truth f and draw samples via von Neumann acceptance-rejection Monte Carlo sampling. The distribution is a negative binomial with parameters given in Figure 7.5 and exponential in Figure 7.6. The number of events N and the Poisson μ were varied, with the histogram bin size Δx kept approximately fixed. One must remember that the overall statistics scales also with the μ-value, so there must be an optimal point with respect to the pileup deterioration and the collected number of events.

The convergence of the iterative algorithms was obtained in every scenario, and the fundamental limitations were hit when the smeared distribution g was almost

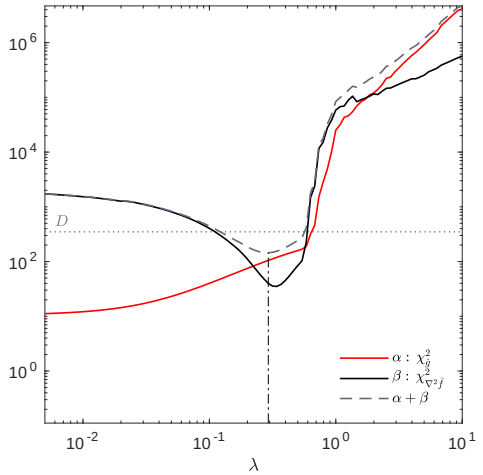

Figure 7.4: Data driven equilibrium between the re-projection cost α and the variational regularity cost β. The horizontal line denotes the number of bins D in the measured histogram $\sim$ number of degrees of freedom.

purely Gaussian, which was the case when $\mu \sim 10$ or more. In that case, the inverted distribution was lacking any fine structure. Frequentist pointwise uncertainty bands ($\alpha = 0.05$) $\sim$ 95 CL were obtained from the bootstrap procedures. The coverage is heuristically reasonably when comparing with respect to the truth, however, the nonlinear oscillation is not completely covered by the bootstrap. A 3σ signal to uncertainty ratio, shown in each figure, seems to be a reasonable safe threshold cut.

In Figure 7.5, we observe some oscillation or 'nucleation' of certain eigenmodes in the solution, which is not an unusual phenomenon in inverse problems. We see that the bias correction seems to increase the variance of the estimates slightly, which is visible in bin-by-bin fluctuations. This is expected due the to bias-variance tradeoff. The optimal number of bias iterations should be studied further, because it is a free parameter unless taken till convergence. In the NBD simulation case, the bias correction seems to have amplified slightly the oscillation via recursion. Here, we used three iterations. Thus, the bias iteration can behave as an additional regulator of the problem. The size of the bootstrap samples for the daughter and mother algorithms should be taken as large as computational resources allow. As a practical guideline, extensive simulations should be always used to investigate the problem-specific stability, regularization, bias-variance and uniqueness.

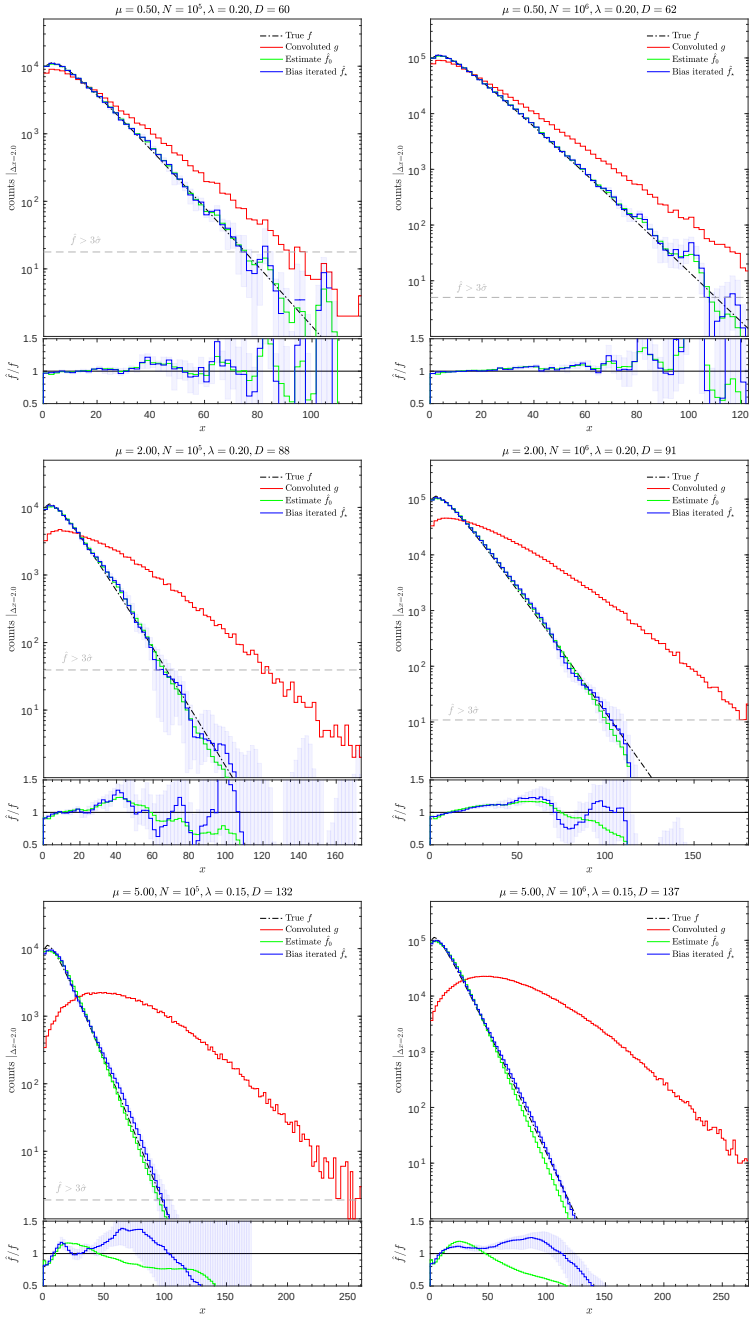

Figure 7.5: Simulations with $f(x) = P_{\text{NBD}}(x; \langle n \rangle = 12.5, k = 1.4)$. The bootstrap based uncertainties are shown in blue.

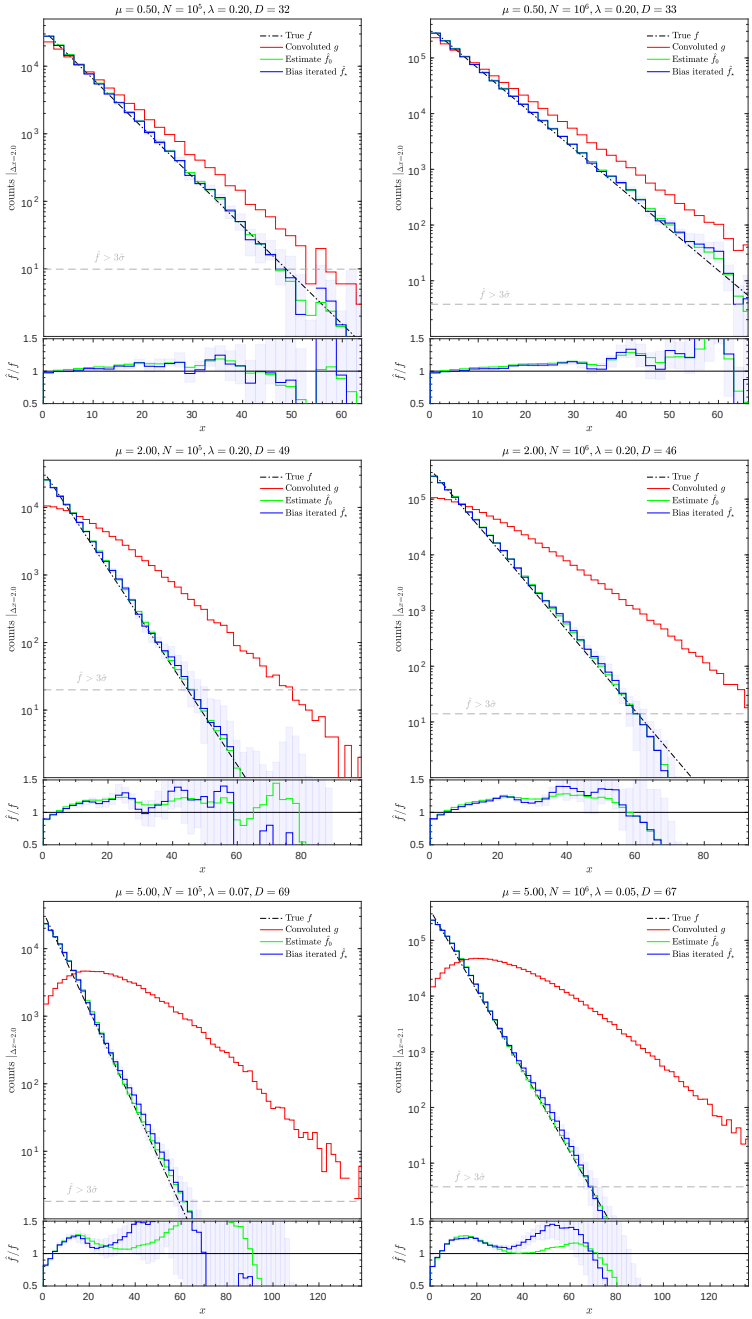

Figure 7.6: Simulations with $f(x) = 1/\alpha \exp(-x/\alpha)$ with $\alpha = 6$. The bootstrap based uncertainties are shown in blue.

7.6 LHC data inversion

For this proof-of-concept study, we use ALICE proton-proton charged particle multiplicity spectra measured at $\sqrt{s} = 0.9$ and 7 TeV from HEPData [226]. The publication includes also data at $\sqrt{s} = 8$ TeV, which we leave out here because of very similarity with $\sqrt{s} = 7$ TeV, but we include it in our analysis code available online. Phenomenologically, the average central multiplicity density $dN_{ch}/d\eta$ in proton-proton collisions follows Regge like power law scaling $\propto s^{\alpha(0)-1}$ with the effective pomeron intercept $\alpha(0) \simeq 1.1$. The event selection definitions are: the INEL (minimum bias inelastic) which in this case means minimal activity in any of the trigger subdetectors and the NSD (non-single-diffractive), which is event rapidity topology based selection. The idea behind the NSD is to suppress *qualitatively* single diffraction events within forward system mass range which result in a large enough pseudorapidity gap on forward or backward pseudorapidity side of the detector. The single diffractive suppression is done simply by requiring event activity on both forward and backward triggers. In addition, the INEL events may have either $N_{ch} \geq 0$ or ≥ 1 particles (tracklets) required in the central region $|\eta| < 1$. We use the $N_{ch} \geq 0$ class, because diffractive events can be with $N_{ch} = 0$ at central but trigger forward detectors.

We executed Kisu inversion simply based on the Poisson compounding hypothesis with different μ values and the regularization being fixed to small $\lambda = 0.03$ in order not to bias the distribution shapes. Varying the regularization strength results in principle in a systematic uncertainty, but we observed it mainly affects the small scale structure. The ALICE data uncertainties are a combination of systematic factors with technically unknown distribution coverage, for example, soft QCD Monte Carlo modeling affecting estimated trigger efficiencies and detector unfolding, and statistical counting fluctuations. For the systematic shape variations we did a minimum and maximum global shape shift and evaluated statistical bin-by-bin bootstrap Poisson re-sampling on top of that, with the given event sample sizes $\mathcal{O}(7 \cdot 10^6)$ at 0.9 TeV and $\mathcal{O}(6 \cdot 10^7)$ at 7 TeV from [226]. The uncertainties in Figures 7.7 and 7.8 represent the 95CL values of this procedure. In principle, one could have done also bin-by-bin bootstrap for the systematic shapes, but because that procedure would neglect all bin-to-bin continuity correlations, it would give way too large high frequency fluctuations not reflecting typical systematic spectrum distortion or bias variations.

As a surprise from the inversion, a strong hidden secondary NBD like peak structure appears at certain μ hypothesis, most strongly with Poisson $\mu \approx 3 \ldots 4$ number of simultaneous 'mini-collisions', with the zero-suppressed average given by Equation

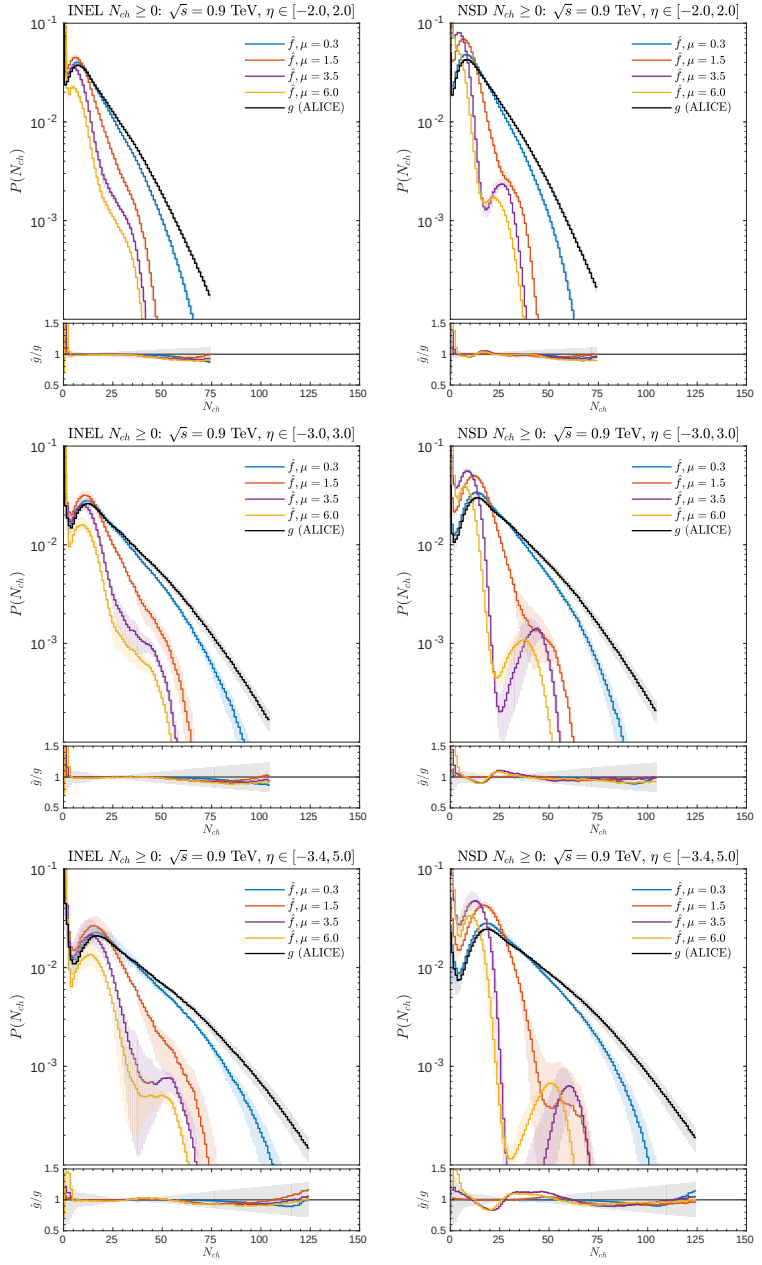

Figure 7.7: ALICE INEL $\sqrt{s} = 0.9$ TeV data on the left and NSD data on the right [226], for three different pseudorapidity intervals and the Kisu inversion results under different μ-hypothesis.

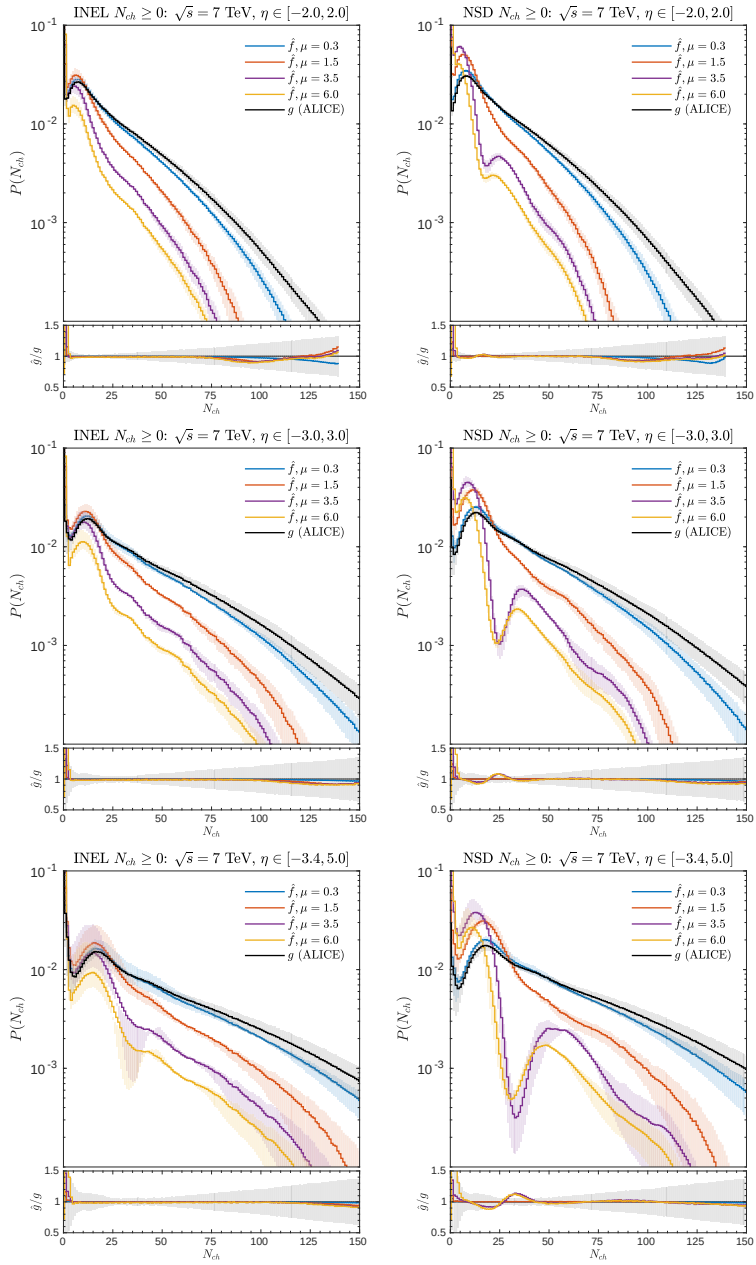

Figure 7.8: ALICE INEL $\sqrt{s} = 7$ TeV data on the left and NSD data on the right [226], for three different pseudorapidity intervals and the Kisu inversion results under different μ-hypothesis.

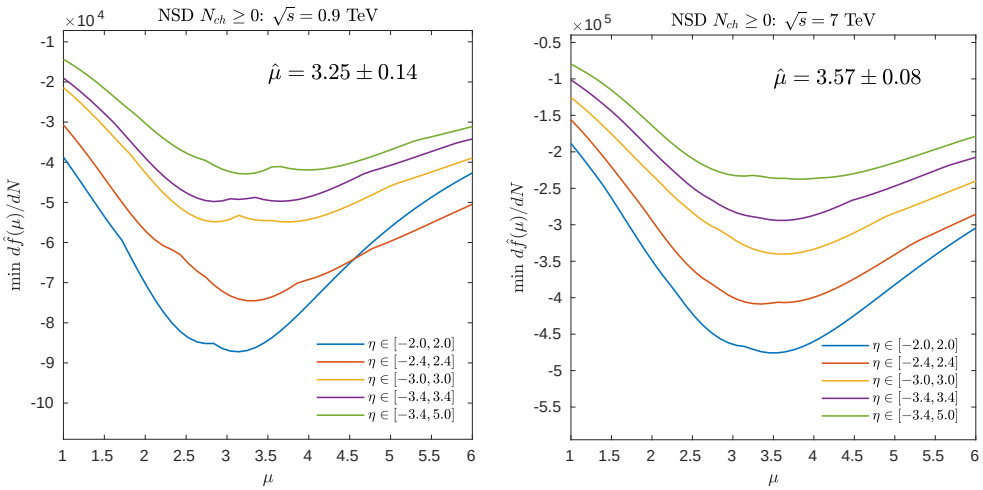

Figure 7.9: ALICE NSD data parameter μ-scans for two different energies, at $\sqrt{s} =$ 0.9 TeV on the left and at 7 TeV on the right.

7.3. Also interesting is that this peak is not strongly visible in the INEL class but appears only once single diffractive events have been suppressed using the NSD class. It seems that the first NBD like peak is similar at both energies, but the second runs with energy. The INEL with $N_{ch} \geq 1$ event class (figures available using our code) gives results roughly similar to the NSD class, this requirement effectively also suppress diffraction. Also, in general the secondary peak increases in relative amplitude when the pseudorapidity window is enlarged and also the peak position shifts. A thorough interpretation of this observation requires further studies. A typical interpretation of the negative binomial double peak structures is that there are two separate mechanisms for the particle production initiator, soft confining and hard point like. The high multiplicity tail is also interesting, usually difficult to describe perfectly by many Monte Carlo models. For comparisons, see [226].

In Figure 7.9, we scan numerically over μ for the minimum of $\frac{d\hat{f}(\mu)}{dN}$ as the criteria for the steepest dip. The results for $\hat{\mu}$ show the mean and its standard error over different pseudorapidity intervals, value growing with energy, which is expected in the multiparton interaction picture with increasing particle densities. At lower energy, there seems to be larger variation with the solutions. We found out that using larger regularization λ values pushes the minimum towards lower values of μ, perhaps simply (over)-smoothing the solutions. Finally, we see in Figure 7.7 and 7.8 ratio plots

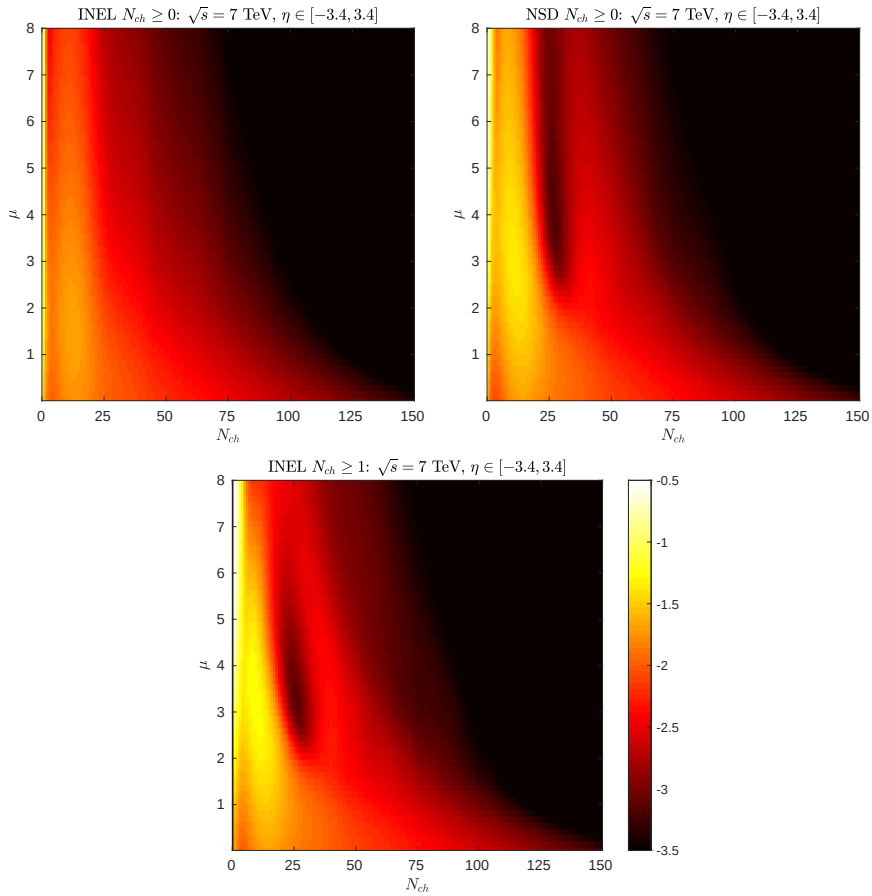

Figure 7.10: Inversion results of $\hat{f}$ in the (N_{ch}, μ) plane at $\sqrt{s} = 7$ TeV for $\eta \in [-3.4, 3.4]$. Colors denote values of $\log_{10}[P(N_{ch})]$.

that the re-projection $\hat{g} = \hat{\mathcal{F}}(\hat{f})$ versus the measured g has the largest tension at very low multiplicities $N_{ch} < 5$, in the domain dominated by diffraction with qualitatively different behavior than the NBD peaks. The second interesting domain is around $N_{ch} \sim 25$, where oscillations are manifest. Figure 7.10 demonstrates the 'topological' differences between the inversion results as a function of the μ parameter. The distribution splits and develops the secondary peak at a certain μ-value in the case of the NSD $N_{ch} \geq 0$ event class. In the case of the INEL $N_{ch} \geq 1$ splitting also happens, and the re-fusion of two peaks happens faster as a function of the growing μ-value. For the INEL $N_{ch} \geq 0$ class, on the other hand, no splitting is visible.

7.7 Conclusions and prospects

We studied the inversion of a stochastic autoconvolution integral equation and showed that useful inverse solutions can be obtained with a novel algorithm. Special effort was invested in the algorithmic construction from bottom-up and in statistical and systematic uncertainty estimation. A new class of semi-model free phenomenological interpretations and statistical tests of inclusive distributions in soft QCD are possible with the algorithm. It is possibly also useful in jet substructure studies when used together with jet algorithms. The algorithm can be also utilized directly as an experimental pileup inversion algorithm in minimum bias studies. There are also possibilities to combine the statistical approach derived here with event-by-event pileup correction algorithms such as [179, 227]. Future directions may include more detailed analytic treatment in terms of both phenomenology of soft QCD, but also technical development of the algorithm for other types of statistical integral equations. The ultimate goal could be higher dimensional tomography of high energy proton, which may be realistic with the help of deep neural network techniques.

Inverting the final state multiplicity spectrum caused by independent multiparton interactions (MPI) is an obvious research target. We did a simple one free parameter inversion using ALICE multiplicity measurements, which exposed an interesting secondary peak structure most prominent with the NSD event selection class. We are not aware of a similar study, typically superposition fits with multiple negative binomial distributions are done, or Monte Carlo model-based studies. One could do a further study where different cms-energies and rapidity intervals are matched together using the autoconvolution picture in a direct or inverse direction. Speaking in the Regge theory language, an approximately similar concept is the multiple independent pomeron exchanges with 'AGK cuts' [25]. Given a model of exchange probabilities of multipomeron exchange with inelastic cuts, such as the inelastic eikonal approximation, in principle we can from the measured observables reconstruct a single pomeron exchange observables without a Monte Carlo model. Deviation from expectations indicates multipomeron (enhanced) interactions which are nonlinear effects not fitting in the K-fold autoconvolution picture. Thus, the inverse algorithm result can be seen as a statistical null-hypothesis generator. Probing the fuzzy interface between multiplicities driven by a semi-hard scale interaction versus soft cut pomerons, could be interesting. Also, the nonlinear gluon saturation conjecture at low Bjorken-x and the robust experimental observables of it are open problems.

Combining the approach here with other mathematical tools, the shadow of diffraction can be studied in a new light. That is, it would be interesting to see

if data can *directly* show evidence for the AGK cuts like phenomena – certain specific additive algebraic rules, multiplicity density, rapidity gap topologies. So far, no water-proof observables with one-to-one mapping with AGK have been measured, as far as we know. We also see that methods here could give new insight to the 'centrality' determination in heavy ion collisions when combined with the Gribov-Glauber model. Inverting the measured transverse energy distribution could lead to a more precise understanding of pure 'binary scaling' (independent nucleon-nucleon collisions) versus strong nuclear effects. The mathematical description here is, essentially, what is approximately meant by binary scaling.

8 DeepEfficiency

We introduce a new high dimensional algorithm for efficiency corrected, maximally Monte Carlo event generator independent fiducial measurements at the LHC and beyond. The approach is driven probabilistically using a Deep Neural Network on an event-by-event basis, trained using detector simulation and even only pure phase space distributed events. This approach gives also a glimpse into the future of high energy physics, where experiments publish new type of measurements in a radically multidimensional way.

Chapter in: arXiv:1809.06101 [physics.data-an]

High energy physics data needs to be corrected for experimental effects induced by the kinematically non-uniform response of the detector and reconstruction algorithms. These corrections are usually dubbed efficiency times acceptance corrections, implemented often by dividing the measured number of background corrected events in a histogram bin with the efficiency value obtained by simulations. In addition, unfolding of histograms of observables is often done to account for distortions of resolution (variance) and absolute value (bias) induced by the detector. Unfolding is usually treated with a stochastic smearing matrix, obtained via simulations. It is the art of unfolding and optimization algorithms, with suitable regularization and possible additional physical boundary conditions, which then "inverts" this response matrix in an algebraic or probabilistic way. An important part is the uncertainty estimation, both in purely statistical terms but also in systematic ways. Well known is that many of the naive bin-by-bin methods are severely biased towards "Monte Carlo truth" [228].

A crucial step of the measurement is the proper definition of the fiducial phase space. Ideally, this should be defined in terms of proper final state observables, such as transverse momentum and pseudorapidity of charged particles, within geometrically visible and electrically active detector volume. A fiducial measurement, by its very definition, minimizes extrapolations of data and thus also minimizes the model dependence. In this work, we go further and propose a *maximally* Monte Carlo event generator independent way to implement fiducial efficiency corrections. For the best of our knowledge, this philosophy and approach what we call *DeepEfficiency*, based on Deep Neural Networks (DNN), is a new one. However, in other contexts the networks are well known in high energy physics, especially in signal-background separation. For a recent review see [229], or for the pioneering studies [230].

For concreteness, let us define our observable degrees of freedom at the level of individual final state particles. Our final state consist of N charged particles (tracks) with 3-momentum $\vec{p}$ of each being measured. This final state spans a real valued vector space $\mathbb{R}^{3N}$, from which one can construct a set of Lorentz scalars and other non-scalar observable distributions, such as transverse momenta or angular ones. The detector has a finite probability $\mathbb{P}$ of seeing the N-track event, given the trigger and tracking efficiencies. This expected probability is represented with an efficiency mapping $\mathcal{E} : \mathbb{R}^{3N} \to \mathbb{P}$. Clearly, by using space-time symmetries of interest, one can often do dimensional reduction. However, treating maximally the abstract detector space, forbids us from doing so. That is, we want to take into account all the correlations that the detector may induce in terms of efficiency losses.

A simplification, dimensional reduction but also a loss of information would occur, if one would for example factorize all tracks and get the full event detection efficiency

as $\mathcal{E}_{tot} = \mathcal{E}_1\mathcal{E}_2 \cdots \mathcal{E}_N$. However, for very high dimensional problems, one may do this. We point out here that the efficiency function $\mathcal{E}$ can be used also within the context of Matrix Element Method (MEM) type likelihood analyses, in addition to the fiducial measurements which are of our interest here. For typical approximations within MEM applications, see [231].

What we simply now do is learn this high dimensional detector efficiency function $\mathcal{E}$ using a fully connected, multilayer DNN in a regression mode. That is, we simply use the neural network as a high dimensional function approximator and interpolator - as a mathematical hammer. The exact architecture, the cost function and its regularization and the gradient descent methods are art forms each of their own, we return to these in extended descriptions. Because we use the fully differential observable kinematic information of the final states, we can even use a pure phase space Monte Carlo event generator as an input to the detector simulation. Perhaps the most intuitive proof of the vanishing dependence on the event generator is based on a multidimensional histogram division picture with hyperbin volumes approaching zero.

A practical requirement for the generator is that a large enough event statistics is produced continuously for all corners of the phase space of interest, naturally. For final states containing jets, a pure phase space Monte Carlo is not feasible. Thus in practice one proceeds as usual with a realistic MC generator of interest. One may do this also in other cases, naturally. Here we do not touch the highly interesting but complex topic of matching optimally parton level, hadron level and detector level objects and sub-structures, which is basically always non-trivial, sometimes ill-posed, with hard or soft QCD processes.

In optimizing (training) the network parameter set Θ, we minimize the so-called cross entropy cost function suitable for our probabilistic inference problem

$$L(\Theta, \mathcal{S}) = -\frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \left[R_i \ln \mathcal{E}_i + (1 - R_i) \ln(1 - \mathcal{E}_i) \right], \tag{8.1}$$

where the sum is over all events in the simulation sample $\mathcal{S}$. The known response R is 0 for efficiency lost events, and 1 for selected events, both within the fiducial phase space. That is, one must not include events outside the fiducial phase space in the sample, by construction. A cutoff regularization is used for the rare $\mathcal{E} \to 0$ singularity. The word entropy comes from the functional form of the cost, related to the Kullback-Leibler [213] divergence which is a difference between cross entropy and entropy, and the KL-divergence itself is related to the Maximum Likelihood principle. In addition, one can add typical regularization schemes such as ℓ_1- (sparsity, Laplace) or ℓ_2-norm (smoothness, Gaussian) based, in order to avoid overfitting. This is avoided also

effectively by choosing the minimal network architecture which results in efficiency corrections with minimal bias and variance on simulated samples. Also the training sample size needs to be high enough, given that the deep network architectures can contain $\mathcal{O}(10^4 - 10^6)$ free parameters.

After training, the efficiency inversion of data and arbitrary observables of interest are obtained with

$$\frac{dN}{d\mathcal{O}} = h_{\mathcal{O}}(\{\vec{p}\}) \odot [\mathcal{E}(\{\vec{p}\})]^{-1}, \tag{8.2}$$

where $h_{\mathcal{O}}$ is a probability distribution estimator operator, typically a bin width $\Delta\mathcal{O}$ normalized histogram. That is, the discretization of observables only enters at this point. The point-wise operator $\odot$ is defined as an integral (sum) over the event sample and the weight $\mathcal{E}(\{\vec{p}\})$ is obtained from the neural network. Simply put, event-by-event, one constructs the observable of interest and calls a weighted histogram fill with a weight $\mathcal{E}^{-1}$. Thus, in an extraordinary smooth way, one can efficiency correct *simultaneously* arbitrary number of single or multidimensional histograms of observables using the same weight. For a related mathematical discussion, see the Horvitz-Thompson unbiased estimator [232]. Differential cross sections are obtained in a standard way normalizing by integrated luminosity. A straightforward way to obtain statistical uncertainties is via Efron's bootstrap re-sampling. The possible algorithmic (network) bias should be empirically tested using simulations, observable by observable, which is easily automated.

The performance was numerically evaluated using the full ALICE detector simulation with input being low-mass, low-p_T central exclusive QCD diffraction, which is a prominent "glueball production process" at the LHC, decaying in this case to charged meson final states $(\pi^+\pi^-, K^+K^-)$. We obtained solid results with a 5-layer network with ~ 100 neurons per layer with a 6-dimensional input, using hyperbolic tangent activation functions and a final layer with one sigmoidal output function. The inversion performance in terms of χ^2/bin was close to unity for a variate of one and two dimensional differential distributions, also in the steep tails. TensorFlow 1.9 [233] was used as the computational framework. The event generator independence lemma was observed to hold with Monte Carlo samples driven by different scattering amplitudes, such as photoproduction of $\rho^0 \to \pi^+\pi^-$ versus $\pi^+\pi^-$ production via double Pomeron exchange. A typical efficiency correction bias coming from unknown angular dependence or spin polarization densities, due to uncertain non-perturbative scattering amplitudes for both continuum and resonance production - is a solved problem now. Also the efficiency correction bias, propagating from a priori unknown but parametrized non-perturbative proton elastic form factors and inelastic structure functions driving the system soft transverse momentum distributions, vanishes.

Summarizing, we introduced a formally optimal and generic efficiency inversion algorithm for fiducial measurements. As a reminder, fiducial measurements are measurements minimizing the kinematic-geometrical acceptance extrapolations. However, in certain cases non-fiducial extrapolations are needed, for example the case of a fully angular flat phase space definition requirement for spin polarization studies. Another interesting topic is the high dimensional, continuous unfolding of distributions, which we did not discuss here. However, one can fuse single dimensional matrix unfolding algorithms with the efficiency inversion of DeepEfficiency.

9 Conclusions

The topic of this thesis was high energy diffraction. We obtained numerous *new results*, the most important are as follows.

1. We developed a novel algebraic-probabilistic framework which we called Combinatorial Superstatistics, as a new observable definition of soft QCD inclusive diffraction. In addition, we posed and solved a new combinatorial Poisson superposition problem with the Möbius inversion theorem. Then we applied the developed tools to the proton-proton data measured in ALICE at the cms energy $\sqrt{s} = 13$ TeV. This resulted in the first explicitly high dimensional unfolded fiducial measurement of soft QCD, as far as we know. One could call this a measurement of the Grassmannian manifold encapsulating the partial cross sections. All different finite field $\mathbb{F}_2^N$ vector space subspaces are living on this manifold, and thus they hold very rich information to be understood in detail. Also, we implemented the first simultaneous multidimensional maximum likelihood extraction of diffractive cross sections and the effective Pomeron intercept. Finally, we demonstrated our novel F^* projection algorithm by reconstructing pseudorapidity gap distributions from incomplete data. The results showed an enhancement for the low mass diffraction in comparison with typical minimum bias event generator tunes.

2. For the exclusive or semi-exclusive diffraction, we developed a new Monte Carlo event generator, Graniitti, written in modern C++. It is currently the leading event generator on this topic regarding custom spin-dependent low mass diffraction scattering amplitudes and computational performance. Using our simulations in comparison with the preliminary ATLAS+ALFA data at the cms energy $\sqrt{s} = 13$ TeV, we conjectured that tensor mesons and tensor glueballs could be produced with different spin polarization mixtures of transverse and longitudinal components. Thus, we claim that this explains non-perturbatively the enigmatic 'glueball filter', at least in the case of tensor resonances. A significant computational future problem is, how to calculate ab initio the resonance

couplings for the low mass resonance spectrum, which are currently free parameters. In addition, the proton dissociation probability and proton structure fluctuations are significant open problems, perhaps to be solved by future advances in lattice field theories. Experimentally, we did pioneering studies with the ALICE semi-exclusive central diffraction data regarding the invariant mass and transverse momentum composition and provided highly automated spherical harmonic decomposition code systematically in different Lorentz rest frames for advanced spin analyses, available in our Graniitti engine.

3. For studying multiparton interactions from a new perspective, we developed a novel inversion algorithm Kisu, for K-fold stochastic autoconvolution integral equations. Then we applied the algorithm to the published ALICE charged particle multiplicity data measured with different cms energies and final state pseudorapidity ranges. We obtained strong evidence that the average number of soft interactions per collision is between 3 to 4 and that two different physical mechanisms are driving the multiplicity spectra because the inversion results in a clear separation between two-component distributions. Both results match the phenomenology often implemented in Monte Carlo event generators. The two-component model hypothesis is a classic one, such as a soft confining QCD and hard point like scattering. Our algorithm provides a new approach to attack this topic.

4. For precision fiducial measurements, we introduced the first fully differential detector efficiency inversion algorithm working event-by-event in higher phase space dimensions. This was based on deep neural networks and we called it DeepEfficiency. Effectively, we relied on the universal function approximation lemma of deep multilayer neural networks. In classic mathematics, the corresponding statement is the Stone-Weierstrass theorem which states that every continuous function can be represented with polynomials on a closed interval. A more generic extension of the efficiency inversion problem would be the higher dimensional unfolding of distributions. A general mathematical theory is still lacking for deep neural networks, which we see as the fundamental future problem to be solved. That would give theoretical tools to design and understand the networks beyond the current style of 'empirical mathematics'. Also this algorithm still relies on a reliable detector simulation, which we see as an open problem to be solved in the future. Higher dimensional variants of the data-driven 'tag-and-probe' methods to estimate efficiencies could be possible.

In terms of the big picture, tools from algebraic geometry and topology would be interesting to be studied in the context of high energy diffraction. From the experimental side, the future forward instrumentation should be improved significantly for precision measurements and discoveries in the forward domain, perhaps with the help of novel material innovations from the condensed matter physics.

Bibliography

[1] J. von Neumann. *First Draft of a Report on the EDVAC, 30 June 1945*. Tech. rep. Contract No W, 1945.

[2] R. P. Feynman. "Space-time approach to quantum electrodynamics". In: *Physical Review* 76.6 (1949), 769.

[3] M. Veltman. *SCHOONSCHIP, A CDC 6600 program for symbolic evaluation of algebraic expressions*. Tech. rep. CERN, 1967.

[4] C.-N. Yang and R. L. Mills. "Conservation of isotopic spin and isotopic gauge invariance". In: *Physical review* 96.1 (1954), 191.

[5] P. V. Hough. *Method and means for recognizing complex patterns*. US Patent 3,069,654. Dec. 1962.

[6] S. Wolfram. *Mathematica: a system for doing mathematics by computer*. Addison-Wesley, 1991.

[7] T. J. Berners-Lee. *Information management: A proposal*. Tech. rep. CERN, 1989.

[8] R. Brun and F. Rademakers. "ROOT–an object oriented data analysis framework". In: *Nuclear Instruments and Methods in Physics Research Section A* 389.1-2 (1997), 81–86.

[9] R. P. Feynman. "Simulating physics with computers". In: *International journal of theoretical physics* 21.6 (1982), 467–488.

[10] S. Linnainmaa. "Taylor expansion of the accumulated rounding error". In: *BIT Numerical Mathematics* 16.2 (1976), 146–160.

[11] A. Shayeghi et al. "Matter-wave interference of a native polypeptide". In: (2019). arXiv: 1910.14538 [quant-ph].

[12] F. M. Grimaldi. *Physico-Mathesis De Lumine, Coloribus, Et Iride, Aliisque Adnexis Libri Duo*. 1665.

[13] G. Giacomelli and M. Jacob. "Physics at the CERN-ISR". In: *Physics Reports* 55.1 (1979), 1–132.

[14] A. Astbury et al. *A 4π solid angle detector for the SPS used as a proton-antiproton collider at a centre of mass energy of 540 GeV*. Tech. rep. CERN-SPSC-78-6, 1978.

[15] L. I. S. Group et al. "LEP design report: Vol. I The LEP injector chain". In: (1983).

[16] G. Voss and B. Wiik. "The electron-proton collider HERA". In: *Annual Review of Nuclear and Particle Science* 44.1 (1994), 413–452.

[17] H. T. Edwards. "The Tevatron energy doubler: a superconducting accelerator". In: *Annual Review of Nuclear and Particle Science* 35.1 (1985), 605–660.

[18] M. Harrison, T. Ludlam, and S. Ozaki. "RHIC project overview". In: *NIMA* 499.2-3 (2003), 235–244.

[19] O. Brüning. *LHC Design Report: The LHC Main Ring*. Vol. 1. 3. European Organization for Nuclear Research, 2004.

[20] M. Benedikt. *LHC Design Report: The LHC Injector Chain*. Vol. 3. CERN, 2004.

[21] H. Fritzsch, M. Gell-Mann, and H. Leutwyler. "Advantages of the color octet gluon picture". In: *Physics Letters B* 47.4 (1973), 365–368.

[22] D. J. Gross and F. Wilczek. "Ultraviolet behavior of non-abelian gauge theories". In: *Physical Review Letters* 30.26 (1973), 1343.

[23] H. D. Politzer. "Reliable perturbative results for strong interactions?" In: *Physical Review Letters* 30.26 (1973), 1346.

[24] H. D. Politzer. "Asymptotic freedom: An approach to strong interactions". In: *Physics Reports* 14.4 (1974), 129–180.

[25] V. Abramowsky, V. Gribov, and O. Kancheli. In: *Sov. J. Nucl. Phys* 18.308 (1974), 21.

[26] A. Buckley et al. "Rivet user manual". In: *Computer Physics Communications* 184.12 (2013), 2803–2819.

[27] A. Karneyeu, L. Mijovic, S. Prestel, and P. Z. Skands. "MCPLOTS: a particle physics resource based on volunteer computing". In: *The European Physical Journal C* 74.2 (2014), 2714.

[28] J. C. Maxwell. "VIII. A dynamical theory of the electromagnetic field". In: *Philosophical transactions of the Royal Society of London* 155 (1865), 459–512.

[29] J. D. Jackson. *Classical electrodynamics*. 1999.

[30] A. Sommerfeld. *Partial differential equations in physics*. Vol. 1. Academic press, 1949.

[31] K. Yee. "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media". In: *IEEE Transactions on antennas and propagation* 14.3 (1966), 302–307.

[32] G. Green. *An essay on the application of mathematical analysis to the theories of electricity and magnetism*. Wezäta-Melins Aktiebolag, 1828.

[33] G. Kirchhoff. "Zur theorie der lichtstrahlen". In: *Annalen der Physik* 254.4 (1883), 663–695.

[34] A. Fresnel. "Mémoire sur la diffraction de la lumière". In: *da p. 339 a p. 475: 1 tav. ft; AQ 210* (1819), 339.

[35] A. Sommerfeld. "Mathematical theory of diffraction". In: *Mathematical Theory of Diffraction (1896)*. Springer, 2004, 9–68.

[36] M. Troyer and U.-J. Wiese. "Computational complexity and fundamental limitations to fermionic quantum Monte Carlo simulations". In: *Physical Review Letters* 94.17 (2005), 170201.

[37] T. Karras, S. Laine, and T. Aila. "A style-based generator architecture for generative adversarial networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, 4401–4410.

[38] G. Antchev et al. "Double diffractive cross-section measurement in the forward region at the LHC". In: *Physical Review Letters* 111.26 (2013), 262001.

[39] UA5 Collaboration. "Diffraction dissociation at the CERN pulsed collider at cm energies of 900 and 200 GeV". In: *Z. Phys. C* 33 (1986), 175–185.

[40] ALICE Collaboration. "Performance of the ALICE Experiment at the CERN LHC". In: *International Journal of Modern Physics A* 29.24 (2014), 1430044.

[41] ALICE Collaboration. "Performance of the ALICE VZERO system". In: *Journal of Instrumentation* 8.10 (2013), P10016.

[42] A. V. Tello. "AD, the ALICE diffractive detector". In: *AIP Conference Proceedings*. Vol. 1819. 1. AIP Publishing. 2017, 040020.

[43] V. Manzari et al. "The silicon pixel detector (SPD) for the ALICE experiment". In: *Journal of Physics G: Nuclear and Particle Physics* 30.8 (2004), S1091.

[44] M. Mieskolainen. "Combinatorial Superstatistics for Soft QCD". In: (2019). arXiv: 1910.06279 [hep-ph].

[45] ALICE Collaboration. *ALICE luminosity determination for pp collisions at* $\sqrt{s} = 13$ *TeV*. Tech. rep. June 2016.

[46] T. Adye. "RooUnfold". In: (2011). DOI: 10.5170/CERN-2011-006.

[47] S. Navin. "Diffraction in Pythia". In: (2010). arXiv: 1005.3894 [hep-ph].

[48] P. C. Hansen and D. P. O'Leary. "The use of the L-curve in the regularization of discrete ill-posed problems". In: *SIAM Journal on Scientific Computing* 14.6 (1993), 1487–1503.

[49] G. Antchev et al. "First measurement of elastic, inelastic and total cross-section at $\sqrt{s} = 13$ TeV by TOTEM and overview of cross-section data at LHC energies". In: *The European Physical Journal C* 79.2 (2019), 103.

[50] M. Mieskolainen. "Bayesian Classification of Hadronic Diffraction in the Collider Detector at Fermilab". MSc thesis. Tampere University of Technology, 2014.

[51] J. Welti. "Inelastic, non-diffractive and diffractive proton-proton cross-section measurements at the LHC". PhD thesis. University of Helsinki, 2017.

[52] Z. L. Matthews. "Proton-proton collisions at the Large Hadron Collider's ALICE Experiment: diffraction and high multiplicity". PhD thesis. University of Birmingham, 2011.

[53] M. Mieskolainen. "GRANIITTI: A Monte Carlo Event Generator for High Energy Diffraction". In: (2019). arXiv: 1910.06300 [hep-ph].

[54] G. Aad et al. "Measurement of differential cross sections for single diffractive dissociation in $\sqrt{s} = 8$ TeV pp collisions using the ATLAS ALFA spectrometer". In: *Journal of High Energy Physics* 2020.2 (2020).

[55] A. Donnachie and P. V. Landshoff. "Total cross-sections". In: *Phys. Lett.* B296 (1992), 227–232.

[56] G. Aad et al. "Rapidity gap cross sections measured with the ATLAS detector in pp collisions at $\sqrt{s} = 7$ TeV". In: *The European Physical Journal C* 72.3 (2012), 1926.

[57] S. Coleman. "There are no classical glueballs". In: *Communications in Mathematical Physics* 55.2 (1977), 113–116.

[58] C. J. Morningstar and M. Peardon. "Glueball spectrum from an anisotropic lattice study". In: *Physical Review D* 60.3 (1999), 034509.

[59] Y. Chen et al. "Glueball spectrum and matrix elements on anisotropic lattices". In: *Physical Review D* 73.1 (2006), 014516.

[60] K. Hashimoto, C.-I. Tan, and S. Terashima. "Glueball decay in holographic QCD". In: *Physical Review D* 77.8 (2008), 086001.

[61] F. Brünner, D. Parganlija, and A. Rebhan. "Glueball decay rates in the Witten-Sakai-Sugimoto model". In: *Physical Review D* 91.10 (2015), 106002.

[62] V. Mathieu, N. Kochelev, and V. Vento. "The physics of glueballs". In: *International Journal of Modern Physics E* 18.01 (2009), 1–49.

[63] W. Ochs. "The status of glueballs". In: *Journal of Physics G: Nuclear and Particle Physics* 40.4 (2013), 043001.

[64] F. E. Close and A. Kirk. "Glueball-qq filter in central hadron production". In: *Physics Letters B* 397.3-4 (1997), 333–338.

[65] L. Baksay et al. "Evidence for double pomeron exchange at the CERN ISR". In: *Physics Letters B* 61.1 (1976), 89–92.

[66] M. Albrow, T. Coughlin, and J. Forshaw. "Central exclusive particle production at high energy hadron colliders". In: *Progress in Particle and Nuclear Physics* 65.2 (2010), 149–184.

[67] J. Adam et al. "Particle identification in ALICE: a Bayesian approach". In: *The European Physical Journal Plus* 131.5 (2016), 168.

[68] J. Neyman and E. S. Pearson. "IX. On the problem of the most efficient tests of statistical hypotheses". In: *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character* 231.694-706 (1933), 289–337.

[69] D. J. Rezende and S. Mohamed. "Variational inference with normalizing flows". In: *arXiv preprint arXiv:1505.05770* (2015).

[70] J. Alme et al. "The ALICE TPC, a large 3-dimensional tracking device with fast readout for ultra-high multiplicity events". In: *Nuclear Instruments and Methods in Physics Research Section A* 622.1 (2010), 316–367.

[71] ALICE Collaboration. "Alignment of the ALICE Inner Tracking System with cosmic-ray tracks". In: *Journal of Instrumentation* 5.03 (2010), P03003.

[72] A. Akindinov et al. "Performance of the ALICE Time-Of-Flight detector at the LHC". In: *The European Physical Journal Plus* 128.4 (2013), 44.

[73] H. Alvensleben et al. "Precise Determination of ρ-ω Interference Parameters from Photoproduction of Vector Mesons Off Nucleon and Nuclei". In: *Physical Review Letters* 27.13 (1971), 888.

[74] T. Regge. "Introduction to complex orbital momenta". In: *Il Nuovo Cimento (1955-1965)* 14.5 (1959), 951–976.

[75] V. Gribov. "A Reggeon diagram technique". In: *Sov. Phys. JETP* 26.414-422 (1968), 27.

[76] S. Mandelstam. "Cuts in the angular-momentum plane-I". In: *Il Nuovo Cimento (1955-1965)* 30.4 (1963), 1127–1147.

[77] V. S. Fadin, E. Kuraev, and L. Lipatov. "On the Pomeranchuk singularity in asymptotically free theories". In: *Physics Letters B* 60.1 (1975), 50–52.

[78] R. C. Brower, J. Polchinski, M. J. Strassler, and C.-I. Tan. "The Pomeron and gauge/string duality". In: *Journal of High Energy Physics* 2007.12 (2007), 005.

[79] L. Susskind. "Strings, black holes, and Lorentz contraction". In: *Physical Review D* 49.12 (1994), 6606.

[80] T. Sjöstrand, S. Mrenna, and P. Skands. "A brief introduction to PYTHIA 8.1". In: *Computer Physics Communications* 178.11 (2008), 852–867.

[81] V. A. Khoze, A. D. Martin, and M. Ryskin. "Soft diffraction and the elastic slope at Tevatron and LHC energies: a multi-Pomeron approach". In: *The European Physical Journal C-Particles and Fields* 18.1 (2000), 167–179.

[82] S. Donnachie, G. Dosch, P. Landshoff, and O. Nachtmann. *Pomeron physics and QCD*. Vol. 19. Cambridge University Press, 2002.

[83] G. Antchev et al. "Proton-proton elastic scattering at the LHC energy of $\sqrt{s} = 7$ TeV". In: *EPL (Europhysics Letters)* 95.4 (2011), 41001.

[84] F. Abe et al. "Measurement of small angle antiproton-proton elastic scattering at $\sqrt{s} = 546$ and 1800 GeV". In: *Physical Review D* 50.9 (1994), 5518.

[85] M. Bozzo et al. "Elastic scattering at the CERN SPS collider up to a four-momentum transfer of 1.55 GeV^2". In: *Physics Letters B* 155.3 (1985), 197–202.

[86] P. V. Landshoff. "Model for elastic scattering at wide angle". In: *Physical Review D* 10.3 (1974), 1024.

[87] A. Donnachie and P. V. Landshoff. "Elastic scattering and diffraction dissociation". In: *Nucl. Phys.* 244.DAMTP 84/6 (1984), 322.

[88] M. Tanabashi et al. "Review of particle physics". In: *Physical Review D* 98.3 (2018), 030001.

[89] L. Harland-Lang, V. Khoze, and M. Ryskin. "Modelling exclusive meson pair production at hadron colliders". In: *The European Physical Journal C* 74.4 (2014), 2848.

[90] L. Harland-Lang, V. Khoze, and M. Ryskin. "Exclusive LHC physics with heavy ions: SuperChic 3". In: *The European Physical Journal C* 79.1 (2019), 39.

[91] B. Schenke, P. Tribedy, and R. Venugopalan. "Event-by-event gluon multiplicity, energy density, and eccentricities in ultrarelativistic heavy-ion collisions". In: *Physical Review C* 86.3 (2012), 034908.

[92] H. Mäntysaari and B. Schenke. "Evidence of strong proton shape fluctuations from incoherent diffraction". In: *Physical Review Letters* 117.5 (2016), 052301.

[93] G. Aad et al. "Measurement of exclusive $\gamma\gamma \to l^+l^-$ production in proton-proton collisions at $\sqrt{s} = 7$ TeV with the ATLAS detector". In: *Physics Letters. B* 749 (2015), 242–261.

[94] E. S. Bols. "Proton-Proton Central Exclusive Pion Production at $\sqrt{s} = 13$ TeV with the ALFA and ATLAS Detector". MSc thesis. University of Copenhagen, Sept. 2017.

[95] M. Aaboud et al. "Measurement of the exclusive $\gamma\gamma \to \mu^+\mu^-$ process in proton–proton collisions at $\sqrt{s} = 13$ TeV with the ATLAS detector". In: *Physics Letters B* 777 (2018), 303–323.

[96] A. Abulencia et al. "Observation of exclusive electron-positron production in hadron-hadron collisions". In: *Physical Review Letters* 98.11 (2007), 112001.

[97] T. Aaltonen et al. "Observation of Exclusive γ γ Production in $p\bar{p}$ Collisions at $\sqrt{s} = 1.96$ TeV". In: *Physical Review Letters* 108.8 (2012), 081801.

[98] T. Aaltonen et al. "Measurement of central exclusive $\pi^+\pi^-$ production in $p\bar{p}$ collisions at $\sqrt{s} = 0.9$ and 1.96 TeV at CDF". In: *Physical Review D* 91.9 (2015), 091101.

[99] S. Chatrchyan et al. "Exclusive diphoton production of dimuon pairs in proton-proton collisions at $\sqrt{s} = 7$ TeV". In: *Journal of High Energy Physics* 1 (2012), 052.

[100] CMS Collaboration. "Exclusive and semi-exclusive $\pi^+\pi^-$ production in proton-proton collisions at $\sqrt{s} = 7$ TeV". In: (2017). arXiv: 1706.08310 [hep-ex].

[101] CMS Collaboration. *Measurement of total and differential cross sections of central exclusive $\pi^+\pi^-$ production in proton-proton collisions at* 5.02 *and* 13 TeV. Tech. rep. 2019.

[102] J. Armitage et al. "Diffraction dissociation in proton-proton collisions at ISR energies". In: *Nuclear Physics B* 194.3 (1982), 365–372.

[103] D. Bernard et al. "The cross section of diffraction dissociation at the CERN SPS collider". In: *Physics Letters B* 186.2 (1987), 227–232.

[104] N. A. Amos et al. "Diffraction dissociation in $\bar{p}p$ collisions at $\sqrt{s} = 1.8$ TeV". In: *Physics Letters B* 301.2-3 (1993), 313–316.

[105] F. Abe et al. "Measurement of $p\bar{p}$ single diffraction dissociation at $\sqrt{s} = 546$ and 1800 GeV". In: *Physical Review D* 50.9 (1994), 5535.

[106] B. Abelev et al. "Measurement of inelastic, single-and double-diffraction cross sections in proton–proton collisions at the LHC with ALICE". In: *The European Physical Journal C* 73.6 (2013), 2456.

[107] V. Khachatryan et al. "Measurement of diffractive dissociation cross sections in pp collisions at $\sqrt{s} = 7$ TeV". In: *Physical Review D* 92.1 (2015), 012003.

[108] J. Pumplin. "Eikonal models for diffraction dissociation on nuclei". In: *Physical Review D* 8.9 (1973), 2899.

[109] A. Donnachie and P. V. Landshoff. "Proton structure function at small Q2". In: *Zeitschrift für Physik C Particles and Fields* 61.1 (1994), 139–145.

[110] P. Lebiedowicz and A. Szczurek. "Exclusive $pp \to pp\pi^+\pi^-$ reaction: From the threshold to LHC". In: *Physical Review D* 81.3 (2010), 036003.

[111] A. Kaidalov, V. Khoze, A. Martin, and M. Ryskin. "Central exclusive diffractive production as a spin-parity analyser: From Hadrons to Higgs". In: *The European Physical Journal C-Particles and Fields* 31.3 (2003), 387–396.

[112] R. Fiore, L. Jenkovszky, and R. Schicker. "Exclusive diffractive resonance production in proton–proton collisions at high energies". In: *The European Physical Journal C* 78.6 (2018), 468.

[113] K. Golec-Biernat and M. Wüsthoff. "Saturation effects in deep inelastic scattering at low Q^2 and its implications on diffraction". In: *Physical Review D* 59.1 (1998), 014017.

[114] S. R. Klein et al. "STARlight: A Monte Carlo simulation program for ultra-peripheral collisions of relativistic ions". In: *Computer Physics Communications* 212 (2017), 258–268.

[115] M. Łuszczak, W. Schäfer, and A. Szczurek. "Production of W^+W^- pairs via $\gamma^*\,\gamma^* \to W^+W^-$ subprocess with photon transverse momenta". In: *Journal of High Energy Physics* 2018.5 (2018), 64.

[116] V. Budnev, I. Ginzburg, G. Meledin, and V. Serbo. "The two-photon particle production mechanism. Physical problems. Applications. Equivalent photon approximation". In: *Physics Reports* 15.4 (1975), 181–282.

[117] F. Ernst, R. Sachs, and K. Wali. "Electromagnetic form factors of the nucleon". In: *Physical Review* 119.3 (1960), 1105.

[118] J. Kelly. "Simple parametrization of nucleon form factors". In: *Physical Review C* 70.6 (2004), 068202.

[119] A. Buckley et al. "LHAPDF6: parton density access in the LHC precision era". In: *The European Physical Journal C* 75.3 (2015), 132.

[120] G. Boroun. "Longitudinal structure function from logarithmic slopes of F_2 at low x". In: *Physical Review C* 97.1 (2018), 015206.

[121] J. Alwall et al. "MadGraph 5: going beyond". In: *Journal of High Energy Physics* 2011.6 (2011), 128.

[122] V. A. Khoze, A. D. Martin, and M. Ryskin. "The rapidity gap Higgs signal at LHC". In: *Physics Letters B* 401.3-4 (1997), 330–336.

[123] J. R. Forshaw. "Diffractive Higgs production: theory". In: (2005). arXiv: hep-ph/0508274 [hep-ph].

[124] T. Coughlin and J. Forshaw. "Central exclusive production in QCD". In: *Journal of High Energy Physics* 2010.1 (2010), 121.

[125] R. E. Cutkosky. "Singularities and discontinuities of Feynman amplitudes". In: *Journal of Mathematical Physics* 1.5 (1960), 429–433.

[126] L. Harland-Lang, V. Khoze, M. Ryskin, and W. Stirling. "Central exclusive production within the Durham model: a review". In: *International Journal of Modern Physics A* 29.17 (2014), 1430031.

[127] M. Kimber, A. D. Martin, and M. Ryskin. "Unintegrated parton distributions and prompt photon hadroproduction". In: *The European Physical Journal C-Particles and Fields* 12.4 (2000), 655–661.

[128] L. Harland-Lang. "Simple form for the low-x generalized parton distributions in the skewed regime". In: *Physical Review D* 88.3 (2013), 034029.

[129] C. Ewerz, M. Maniatis, and O. Nachtmann. "A model for soft high-energy scattering: tensor pomeron and vector odderon". In: *Annals of Physics* 342 (2014), 31–77.

[130] I. Iatrakis, A. Ramamurti, and E. Shuryak. "Pomeron interactions from the Einstein-Hilbert action". In: *Physical Review D* 94.4 (2016), 045005.

[131] P. Lebiedowicz, O. Nachtmann, and A. Szczurek. "Central exclusive diffractive production of the $\pi^+\pi^-$ continuum, scalar, and tensor resonances in pp and $p\bar{p}$ scattering within the tensor Pomeron approach". In: *Physical Review D* 93.5 (2016), 054015.

[132] W. Landry. "Implementing a high performance tensor library". In: *Scientific Programming* 11.4 (2003), 273–290.

[133] M. Jacob and G. C. Wick. "On the general theory of collisions for particles with spin". In: *Annals of Physics* 7.4 (1959), 404–428.

[134] E. Leader. *Spin in particle physics*. Vol. 15. Cambridge University Press, 2005.

[135] S.-U. Chung. *Spin Formalisms (updated version)*. Tech. rep. Brookhaven National Laboratory, US, 2006.

[136] C. Amsler and J. Bizot. "Simulation of angular distributions and correlations in the decay of particles with spin". In: *Computer Physics Communications* 30.1 (1983), 21–30.

[137] R. A. Kycia, J. Chwastowski, R. Staszewski, and J. Turnau. "GenEx: A simple generator structure for exclusive processes in high energy collisions". In: *Commun. Comput. Phys.* 24.3 (2018), 860–884.

[138] F. E. James. *Monte Carlo phase space*. Tech. rep. CERN, 1968.

[139] R. Kleiss, S. Ellis, and W. J. Stirling. "A new Monte Carlo treatment of multiparticle phase space at high energies". In: *Comput. Phys. Commun.* 40.CERN-TH-4299-85 (1985), 359–373.

[140] G. P. Lepage. "A new algorithm for adaptive multidimensional integration". In: *Journal of Computational Physics* 27.2 (1978), 192–203.

[141] T. Müller et al. "Neural importance sampling". In: *ACM Transactions on Graphics (TOG)* 38.5 (2019), 145.

[142] C. Lam and W.-K. Tung. "Systematic approach to inclusive lepton pair production in hadronic collisions". In: *Physical Review D* 18.7 (1978), 2447.

[143] J. C. Collins and D. E. Soper. "Angular distribution of dileptons in high-energy hadron collisions". In: *Physical Review D* 16.7 (1977), 2219.

[144] K. Gottfried and J. D. Jackson. "On the connection between production mechanism and decay of resonances at high energies". In: *Il Nuovo Cimento (1955-1965)* 33.2 (1964), 309–330.

[145] R. S. Longacre. *Techniques in Meson Spectroscopy*. Brookhaven National Laboratory, Aug. 1991.

[146] M. Mieskolainen. "DeepEfficiency - optimal efficiency inversion in higher dimensions at the LHC". In: (2018). arXiv: 1809.06101 [physics.data-an].

[147] M. Lüscher. "A portable high-quality random number generator for lattice field theory simulations". In: *Computer physics communications* 79.1 (1994), 100–110.

[148] R. Schicker and ALICE. "Central diffraction in ALICE". In: *AIP Conference Proceedings*. Vol. 1350. 1. AIP. 2011, 107–110.

[149] G.-C. Rota. "On the foundations of combinatorial theory I. Theory of Möbius functions". In: *Probability theory and related fields* 2.4 (1964), 340–368.

[150] N.-x. Chen. "Modified Möbius inverse formula and its applications in physics". In: *Physical Review Letters* 64.11 (1990), 1193.

[151] K. Kowalski, W. N. Polyzou, and E. F. Redish. "Partition combinatorics and multiparticle scattering theory". In: *Journal of Mathematical Physics* 22.9 (1981), 1965–1982.

[152] D. Spector. "Supersymmetry and the Möbius inversion function". In: *Communications in mathematical physics* 127.2 (1990), 239–252.

[153] B. Julia. "Statistical theory of numbers". In: *Number theory and physics*. Springer, 1990, 276–293.

[154] E. Onofri, G. Veneziano, and J. Wosiek. "Supersymmetry and combinatorics". In: *Communications in mathematical physics* 274.2 (2007), 343–355.

[155] ATLAS Collaboration. "Luminosity Determination Using the ATLAS Detector". In: *ATLAS-CONF*. Vol. 20. 2010, 10–060.

[156] K. Oyama, A. Collaboration, et al. "Cross-section normalization in proton–proton collisions at 2.76 and 7 TeV, with ALICE at the LHC". In: *Journal of Physics G: Nuclear and Particle Physics* 38.12 (2011), 124131.

[157] M. Mieskolainen. "On the Inversion of High Energy Proton". In: (2019). arXiv: 1905.12585 [hep-ph].

[158] B. Dai, S. Ding, G. Wahba, et al. "Multivariate Bernoulli distribution". In: *Bernoulli* 19.4 (2013), 1465–1483.

[159] J. L. Teugels. "Some representations of the multivariate Bernoulli and binomial distributions". In: *Journal of multivariate analysis* 32.2 (1990), 256–268.

[160] A. Addazi, M. Bianchi, and G. Veneziano. "Glimpses of black hole formation/evaporation in highly inelastic, ultra-planckian string collisions". In: *Journal of High Energy Physics* 2017.2 (2017), 111.

[161] J. Bartels and M. G. Ryskin. "The Space-time picture of the Wee partons and the AGK cutting rules in perturbative QCD". In: *Zeitschrift für Physik C Particles and Fields* 76.2 (1997), 241–255.

[162] J. Bartels, M. Salvadore, and G. Vacca. "AGK cutting rules and multiple scattering in hadronic collisions". In: *The European Physical Journal C-Particles and Fields* 42.1 (2005), 53–71.

[163] S. Ostapchenko. "Monte Carlo treatment of hadronic interactions in enhanced Pomeron scheme: QGSJET-II model". In: *Physical Review D* 83.1 (2011), 014018.

[164] D. Darling. "On a class of problems related to the random division of an interval". In: *The Annals of Mathematical Statistics* (1953), 239–253.

[165] L. W. Beineke and F. Harary. "The genus of the n-cube". In: *Canadian Journal of Mathematics* 17 (1965), 494–496.

[166] R. P. Stanley. "Enumerative combinatorics". In: *Belmont, CA* (1986).

[167] R. J. Glauber. "The quantum theory of optical coherence". In: *Physical Review* 130.6 (1963), 2529.

[168] K. Goulianos. "Diffractive interactions of hadrons at high energies". In: *Physics Reports* 101.3 (1983), 169–219.

[169] E.-J. Ahn et al. "Cosmic ray interaction event generator SIBYLL 2.1". In: *Physical Review D* 80.9 (2009), 094003.

[170] R. Ciesielski and K. Goulianos. "MBR Monte Carlo Simulation in PYTHIA8". In: (2012). arXiv: 1205.1446 [hep-ph].

[171] T. Sjöstrand et al. "An introduction to PYTHIA 8.2". In: *Computer physics communications* 191 (2015), 159–177.

[172] T. Sjöstrand, S. Mrenna, and P. Skands. "PYTHIA 6.4 physics and manual". In: *Journal of High Energy Physics* 2006.05 (2006), 026.

[173] F. W. Bopp, R. Engel, and J. Ranft. "Rapidity gaps and the PHOJET Monte Carlo". In: (1998). arXiv: hep-ph/9803437 [hep-ph].

[174] T. Pierog et al. "EPOS LHC: Test of collective hadronization with data measured at the CERN Large Hadron Collider". In: *Physical Review C* 92.3 (2015), 034906.

[175] T. Lappi and L. McLerran. "Some features of the glasma". In: *Nuclear Physics A* 772.3-4 (2006), 200–212.

[176] F. Gelis, T. Lappi, and R. Venugopalan. "High energy factorization in nucleus-nucleus collisions. II. Multigluon correlations". In: *Physical Review D* 78.5 (2008), 054020.

[177] E. Wigner. "On the quantum correction for thermodynamic equilibrium". In: *Physical review* 40.5 (1932), 749.

[178] R. P. Feynman. "Negative probability". In: *Quantum implications: essays in honour of David Bohm* (1987), 235–248.

[179] D. Bertolini, P. Harris, M. Low, and N. Tran. "Pileup per particle identification". In: *Journal of High Energy Physics* 2014.10 (2014), 1–22.

[180] A. J. Larkoski, S. Marzani, G. Soyez, and J. Thaler. "Soft drop". In: *Journal of High Energy Physics* 2014.5 (2014), 1–46.

[181] G. Soyez et al. "Pileup subtraction for jet shapes". In: *Physical Review Letters* 110.16 (2013), 162001.

[182] M. Cacciari and G. P. Salam. "Pileup subtraction using jet areas". In: *Physics Letters B* 659.1 (2008), 119–126.

[183] R. Hagedorn. "Statistical thermodynamics of strong interactions at high energies". In: *Nuovo Cimento, Suppl.* 3.CERN-TH-520 (1965), 147–186.

[184] R. P. Feynman. "Very high-energy collisions of hadrons". In: *Physical Review Letters* 23.24 (1969), 1415.

[185] A. Polyakov. "Hypothesis of self-similarity in strong interactions: 1. Plural generation of e+e- annihilation hadrons". In: *Zh. Eksp. Teor. Fiz* 59 (1970), 542.

[186] Z. Koba, H. B. Nielsen, and P. Olesen. "Scaling of multiplicity distributions in high energy hadron collisions". In: *Nuclear Physics B* 40 (1972), 317–334.

[187] X. Artru and G. Mennessier. "String model and multiproduction". In: *Nuclear Physics B* 70.1 (1974), 93–115.

[188] R. D. Field and R. P. Feynman. "A parametrization of the properties of quark jets". In: *Nuclear Physics B* 136.1 (1978), 1–76.

[189] B. Andersson, G. Gustafson, and C. Peterson. "A semiclassical model for quark jet fragmentation". In: *Zeitschrift für Physik C Particles and Fields* 1.1 (1979), 105–116.

[190] B. Andersson, G. Gustafson, G. Ingelman, and T. Sjöstrand. "Parton fragmentation and string dynamics". In: *Physics Reports* 97.2-3 (1983), 31–145.

[191] B. R. Webber. "A QCD model for jet fragmentation including soft gluon interference". In: *Nuclear Physics B* 238.3 (1984), 492–528.

[192] A. Kaidalov. "The quark-gluon structure of the pomeron and the rise of inclusive spectra at high energies". In: *Physics Letters B* 116.6 (1982), 459–463.

[193] Y. I. Azimov, Y. L. Dokshitzer, V. A. Khoze, and S. Trovan. "Similarity of parton and hadron spectra in QCD jets". In: *Zeitschrift für Physik C Particles and Fields* 27.1 (1985), 65–72.

[194] G. Zech. "Analysis of distorted measurements – parameter estimation and unfolding". In: (2016). arXiv: 1607.06910 [physics.data-an].

[195] M. Kuusela and V. M. Panaretos. "Statistical unfolding of elementary particle spectra: Empirical Bayes estimation and bias-corrected uncertainty quantification". In: *The Annals of Applied Statistics* 9.3 (2015), 1671–1705.

[196] V. Blobel. "An Unfolding Method for High Energy Physics Experiments". In: (2002). arXiv: hep-ex/0208022 [hep-ex].

[197] G. D'Agostini. "A multidimensional unfolding method based on Bayes' theorem". In: *Nuclear Instruments and Methods in Physics Research Section A* 362.2 (1995), 487–498.

[198] D. Gerth et al. "Regularization of an autoconvolution problem in ultrashort laser pulse characterization". In: *Inverse Problems in Science and Engineering* 22.2 (2014), 245–266.

[199] R. Gorenflo and B. Hofmann. "On autoconvolution and regularization". In: *Inverse Problems* 10.2 (1994), 353.

[200] K. Choi and A. D. Lanterman. "An iterative deautoconvolution algorithm for nonnegative functions". In: *Inverse problems* 21.3 (2005), 981.

[201] A. Meister. "Optimal convergence rates for density estimation from grouped data". In: *Statistics & probability letters* 77.11 (2007), 1091–1097.

[202] B. Van Es, S. Gugushvili, P. Spreij, et al. "A kernel type nonparametric density estimator for decompounding". In: *Bernoulli* 13.3 (2007), 672–694.

[203] G. Martin, K. O'Bryant, et al. "The supremum of autoconvolutions, with applications to additive number theory". In: *Illinois Journal of Mathematics* 53.1 (2009), 219–235.

[204] S. Moskow and J. C. Schotland. "Numerical studies of the inverse Born series for diffuse waves". In: *Inverse Problems* 25.9 (2009), 095007.

[205] R. W. Grubbström and O. Tang. "The moments and central moments of a compound distribution". In: *European Journal of Operational Research* 170.1 (2006), 106–119.

[206] A. Giovannini and L. Van Hove. "Negative binomial multiplicity distributions in high energy hadron collisions". In: *Zeitschrift für Physik C Particles and Fields* 30.3 (1986), 391–400.

[207] G. Alner et al. "Multiplicity distributions in different pseudorapidity intervals at a CMS energy of 540 GeV". In: *Physics Letters B* 160.1-3 (1985), 193–198.

[208] P. Ghosh. "Negative binomial multiplicity distribution in proton-proton collisions in limited pseudorapidity intervals at LHC up to $\sqrt{s} = 7$ TeV and the clan model". In: *Physical Review D* 85.5 (2012), 054017.

[209] B. Andersson, P. Dahlqvist, and G. Gustafson. "On local parton-hadron duality". In: *Zeitschrift für Physik C Particles and Fields* 44.3 (1989), 461–466.

[210] W. Kittel and E. A. De Wolf. *Soft multihadron dynamics*. World Scientific, 2005.

[211] Y. Dokshitzer, V. Khoze, A. Mueller, and S. Troyan. *Basics of perturbative QCD*. Atlantica Séguier Frontières, 1991.

[212] M. Bøgsted and S. M. Pitts. "Decompounding random sums: a nonparametric approach". In: *Annals of the Institute of Statistical Mathematics* 62.5 (2010), 855–872.

[213] S. Kullback and R. A. Leibler. "On information and sufficiency". In: *The Annals of Mathematical Statistics* 22.1 (1951), 79–86.

[214] I. Csiszar. "Why least squares and maximum entropy? An axiomatic approach to inference for linear inverse problems". In: *The Annals of Statistics* (1991), 2032–2066.

[215] A. P. Dempster, N. M. Laird, and D. B. Rubin. "Maximum likelihood from incomplete data via the EM algorithm". In: *Journal of the royal statistical society. Series B (methodological)* (1977), 1–38.

[216] W. H. Richardson. "Bayesian-based iterative method of image restoration". In: *JOSA* 62.1 (1972), 55–59.

[217] L. B. Lucy. "An iterative technique for the rectification of observed distributions". In: *The astronomical journal* 79 (1974), 745.

[218] J. L. Mueller and S. Siltanen. *Linear and nonlinear inverse problems with practical applications*. Vol. 10. Siam, 2012.

[219] J. M. Bardsley and C. R. Vogel. "A nonnegatively constrained convex programming method for image reconstruction". In: *SIAM Journal on Scientific Computing* 25.4 (2004), 1326–1343.

[220] N. Wiener. *Extrapolation, interpolation, and smoothing of stationary time series*. Vol. 7. MIT press Cambridge, MA, 1949.

[221] L. Landweber. "An iteration formula for Fredholm integral equations of the first kind". In: *American journal of mathematics* 73.3 (1951), 615–624.

[222] B. Efron. "Bootstrap Methods: Another Look at the Jackknife". In: *The Annals of Statistics* (1979), 1–26.

[223] P. Hall. "On the bootstrap and confidence intervals". In: *The Annals of Statistics* (1986), 1431–1452.

[224] P. Hall. *The bootstrap and Edgeworth expansion*. Springer Science & Business Media, 2013.

[225] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. CRC press, 1994.

[226] S. Acharya et al. "Charged-particle multiplicity distributions over a wide pseudorapidity range in proton-proton collisions at $\sqrt{s} = 0.9$, 7, and 8 TeV". In: *Eur. Phys. J.* C77.12 (2017), 852.

[227] M. Cacciari, G. P. Salam, and G. Soyez. "SoftKiller, a particle-level pileup removal method". In: *The European Physical Journal C* 75.2 (2015), 59.

[228] G. Cowan. *Statistical data analysis*. Oxford University Press, 1998.

[229] D. Guest, K. Cranmer, and D. Whiteson. "Deep Learning and its Application to LHC Physics". In: *Annu. Rev. Nucl. Part. Sci.* 68 (2018), 1–22.

[230] B. Denby. "Neural networks and cellular automata in experimental high energy physics". In: *Computer Physics Communications* 49.3 (1988), 429–448.

[231] F. Fiedler et al. "The matrix element method and its application to measurements of the top quark mass". In: *NIMA* 624.1 (2010), 203–218.

[232] D. G. Horvitz and D. J. Thompson. "A generalization of sampling without replacement from a finite universe". In: *Journal of the American Statistical Association* 47.260 (1952), 663–685.

[233] M. Abadi et al. "Tensorflow: a system for large-scale machine learning." In: *OSDI*. Vol. 16. 2016, 265–283.

[234] F. E. Close and G. A. Schuler. "Central production of mesons: Exotic states versus pomeron structure". In: *Physics Letters B* 458.1 (1999), 127–136.

[235] S. Mallat. *A wavelet tour of signal processing*. Elsevier, 1999.

A ALICE measurements

A.1 Detector level marginal distributions

The detectors in Figures A.1-A.13 are from left to right:
ZDN/P$_C$, AD$_C$, V0$_C$, SPD$_C$, SPD$_A$, V0$_A$, AD$_A$, ZDN/P$_A$.

For each signal combination, the detector level charge Q (analog-to-digital conversion units) is shown in the upper row and the signal time t (ns) in the lower row. An exception to this is ZDN/P, where the upper row shows the ZDN charge and the lower row is for the ZDP charge. The general color coding is: white (signal accepted), gray (signal not accepted) and pink (spectator, not used in the analysis). The simulations are run through the full ALICE GEANT simulation with a run anchored calibration.

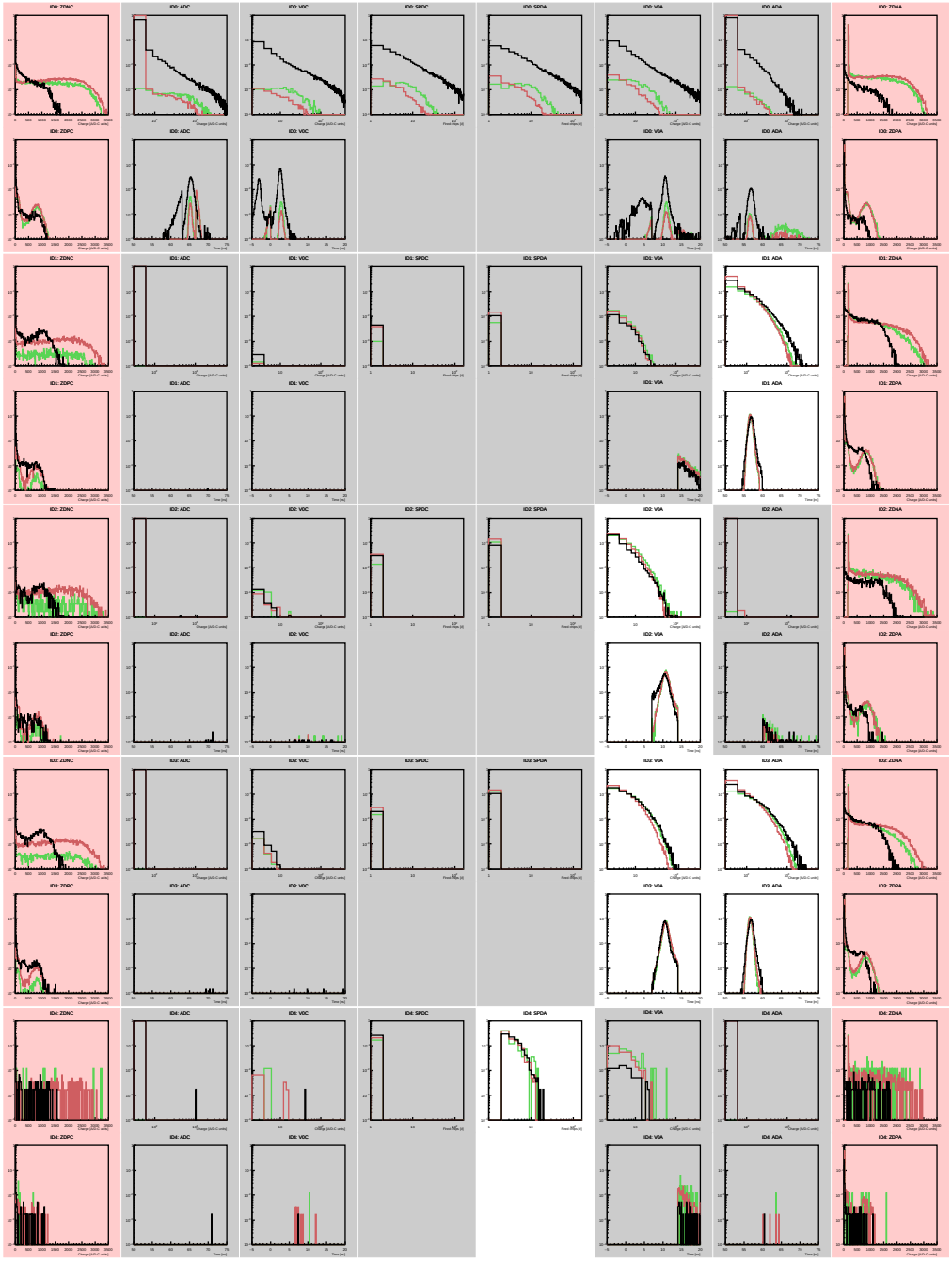

Figure A.1: ALICE **data** at $\sqrt{s} = 13$ TeV (black) and Pythia 6 (red), Phojet (green) simulations for combinations [0-4]: Detector level charge and time distributions.

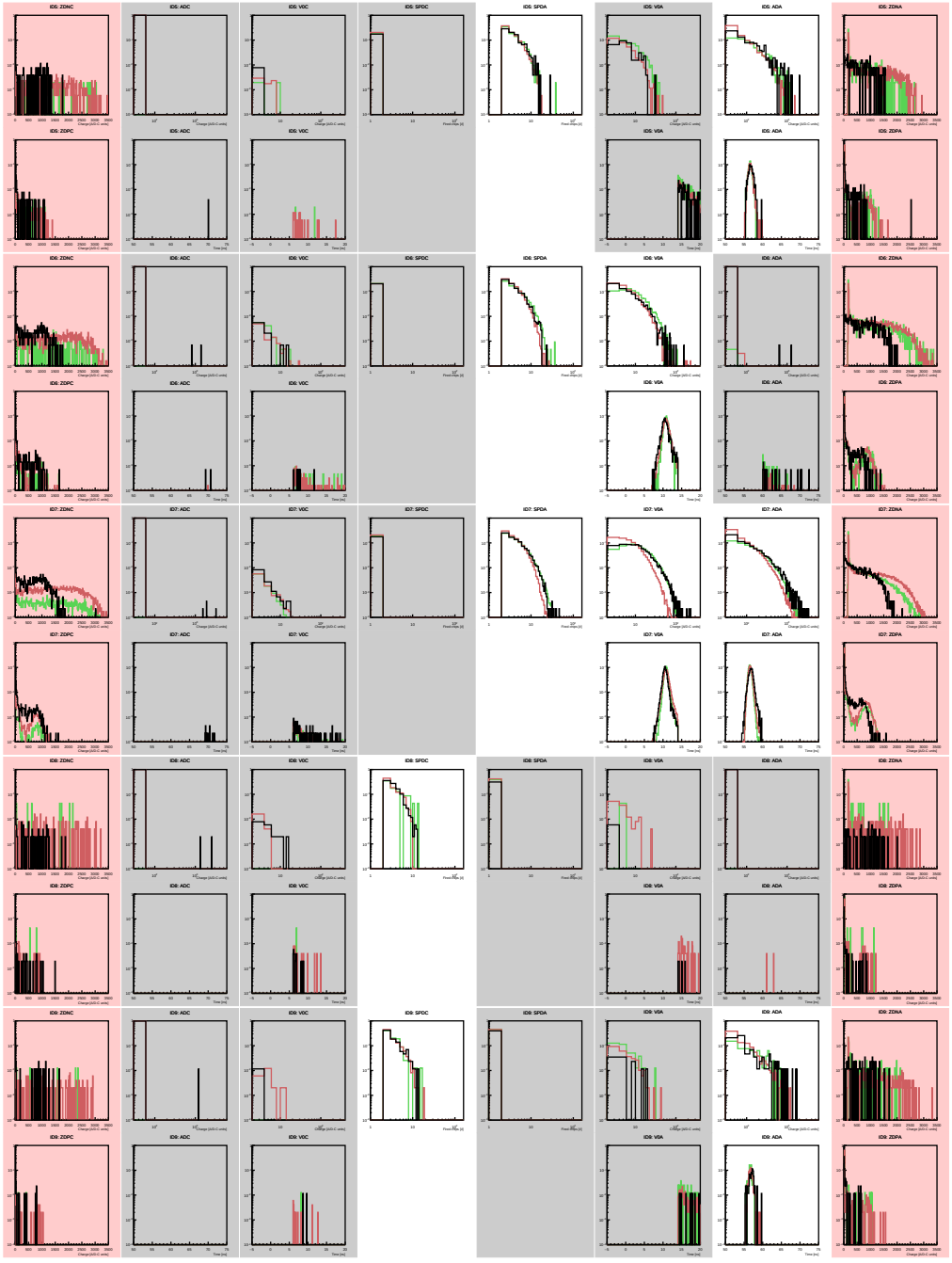

Figure A.2: ALICE **data** at $\sqrt{s} = 13$ TeV (black) and Pythia 6 (red), Phojet (green) simulations for combinations [5-9]: Detector level charge and time distributions.

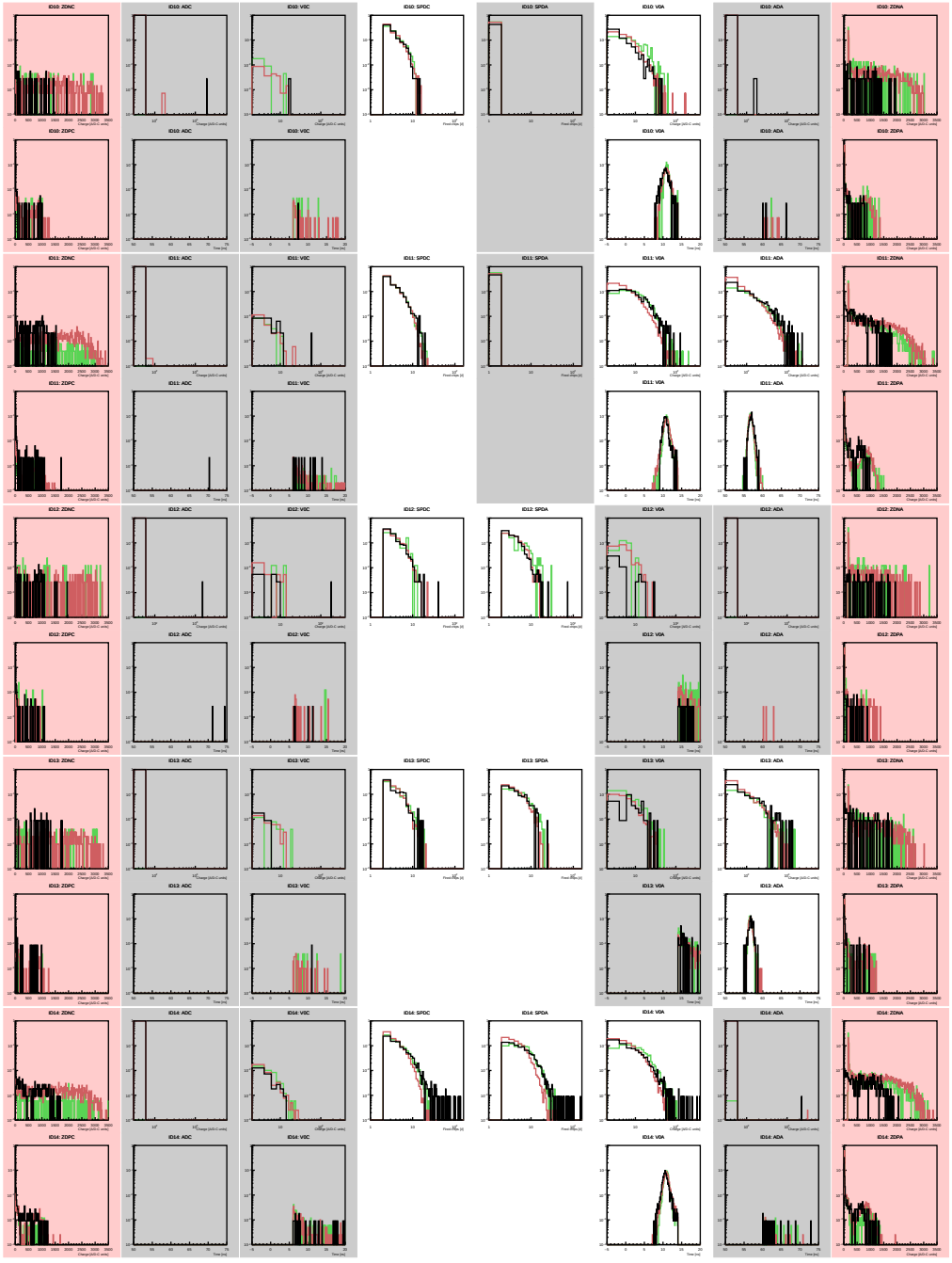

Figure A.3: ALICE **data** at $\sqrt{s} = 13$ TeV (black) and Pythia 6 (red), Phojet (green) simulations for combinations [10-14]: Detector level charge and time distributions.

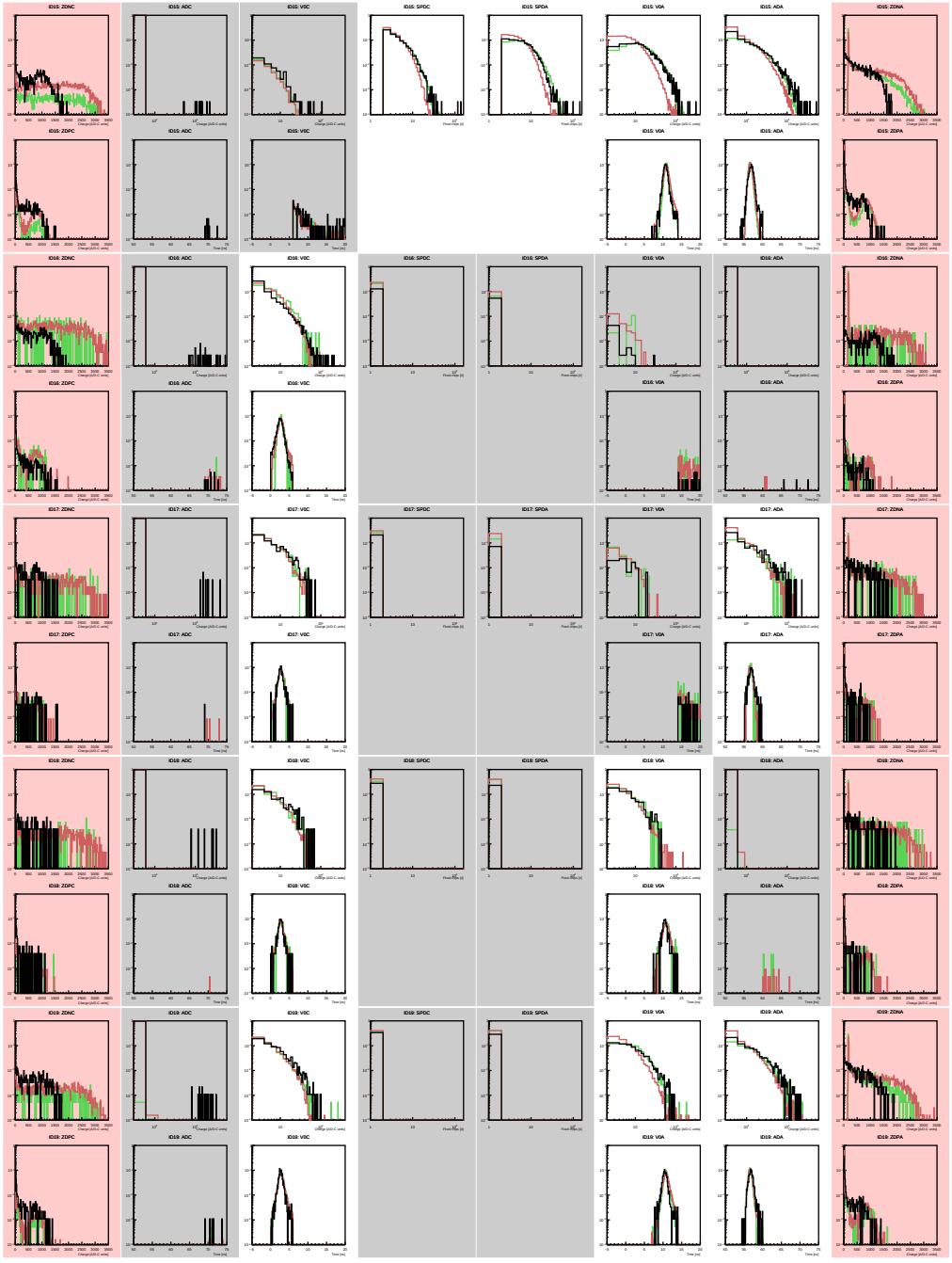

Figure A.4: ALICE **data** at $\sqrt{s} = 13$ TeV (black) and Pythia 6 (red), Phojet (green) simulations for combinations [15-19]: Detector level charge and time distributions.

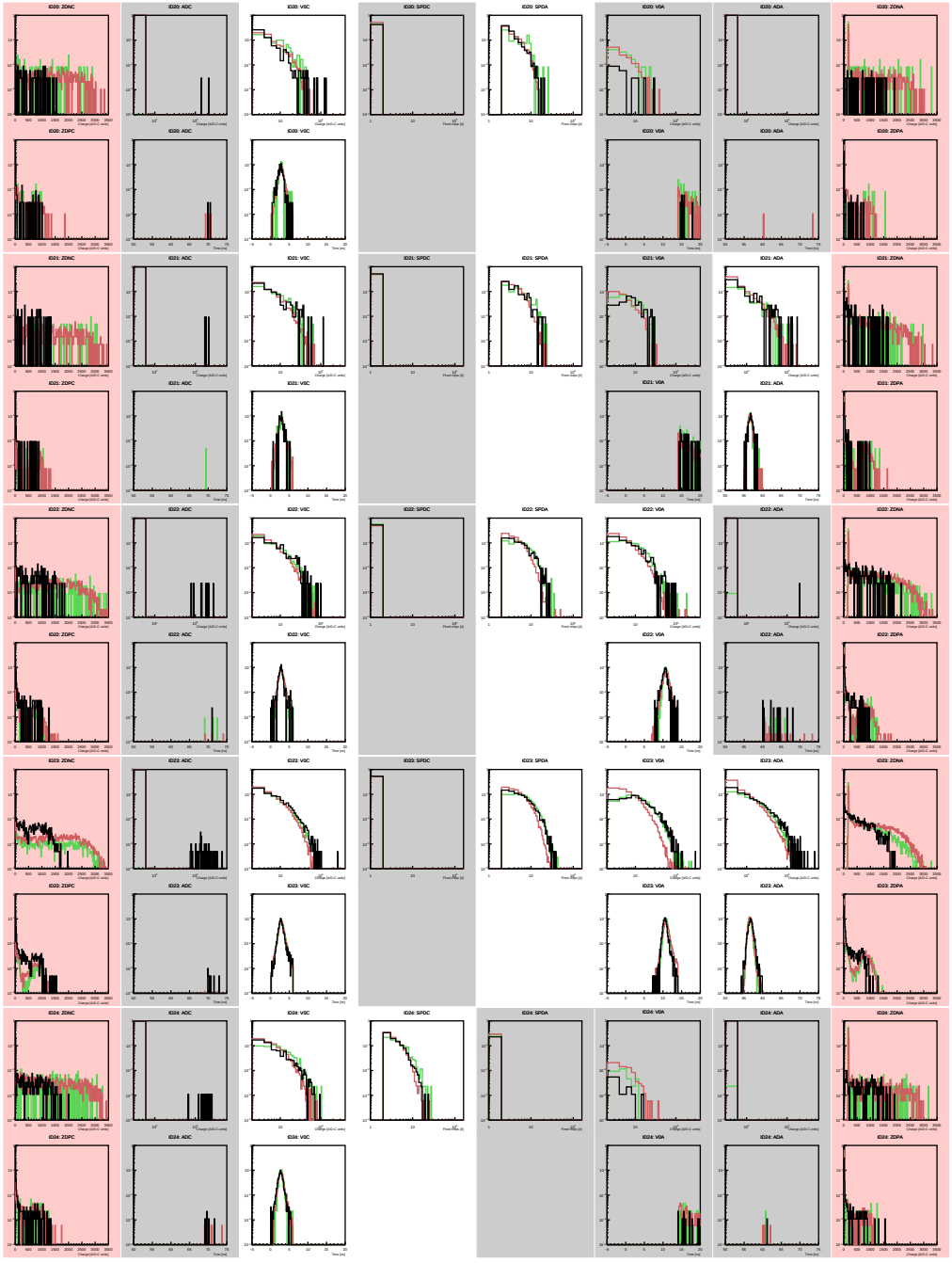

Figure A.5: ALICE **data** at $\sqrt{s} = 13$ TeV (black) and Pythia 6 (red), Phojet (green) simulations for combinations [20-24]: Detector level charge and time distributions.

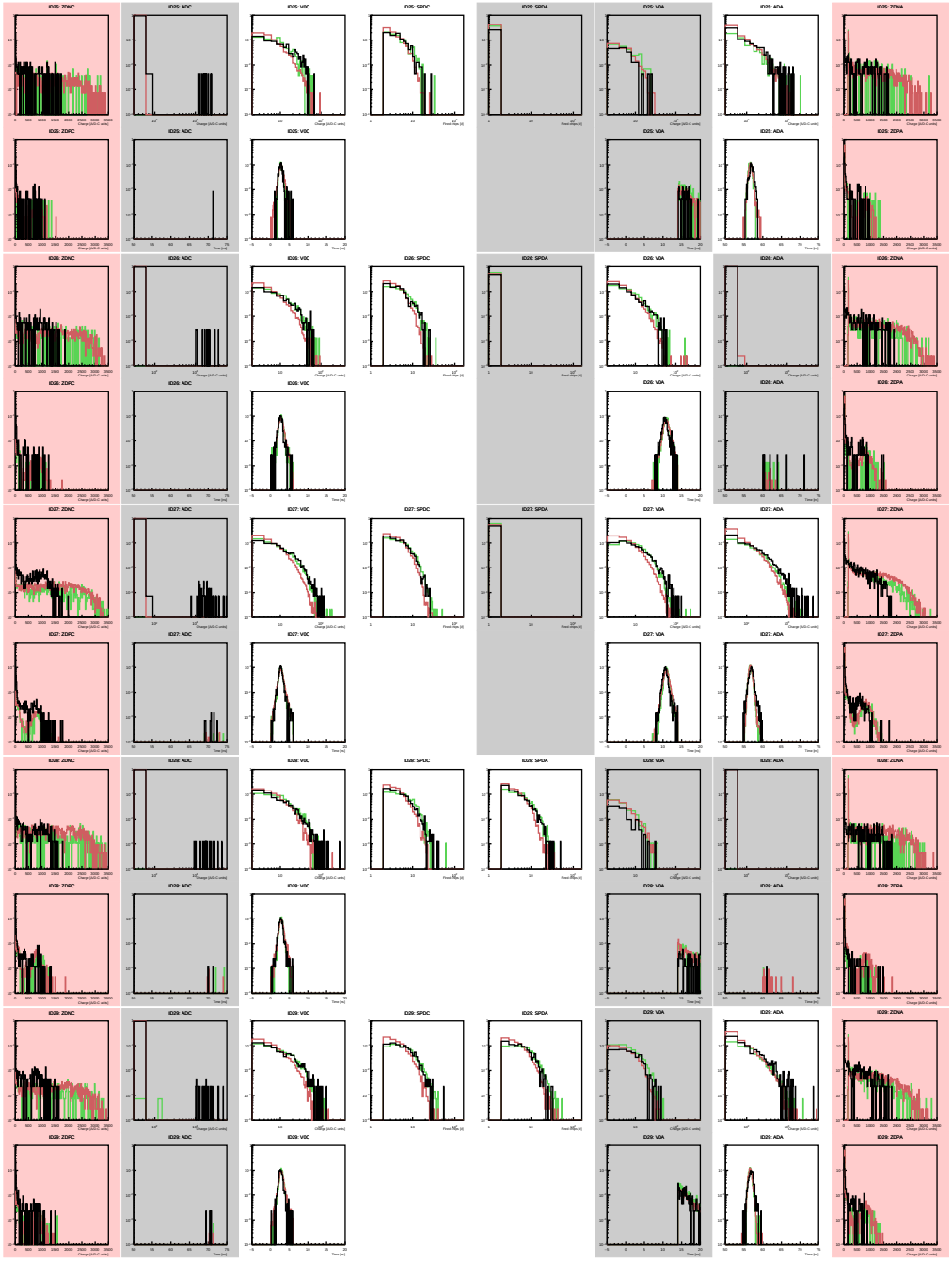

Figure A.6: ALICE **data** at $\sqrt{s} = 13$ TeV (black) and Pythia 6 (red), Phojet (green) simulations for combinations [25-29]: Detector level charge and time distributions.

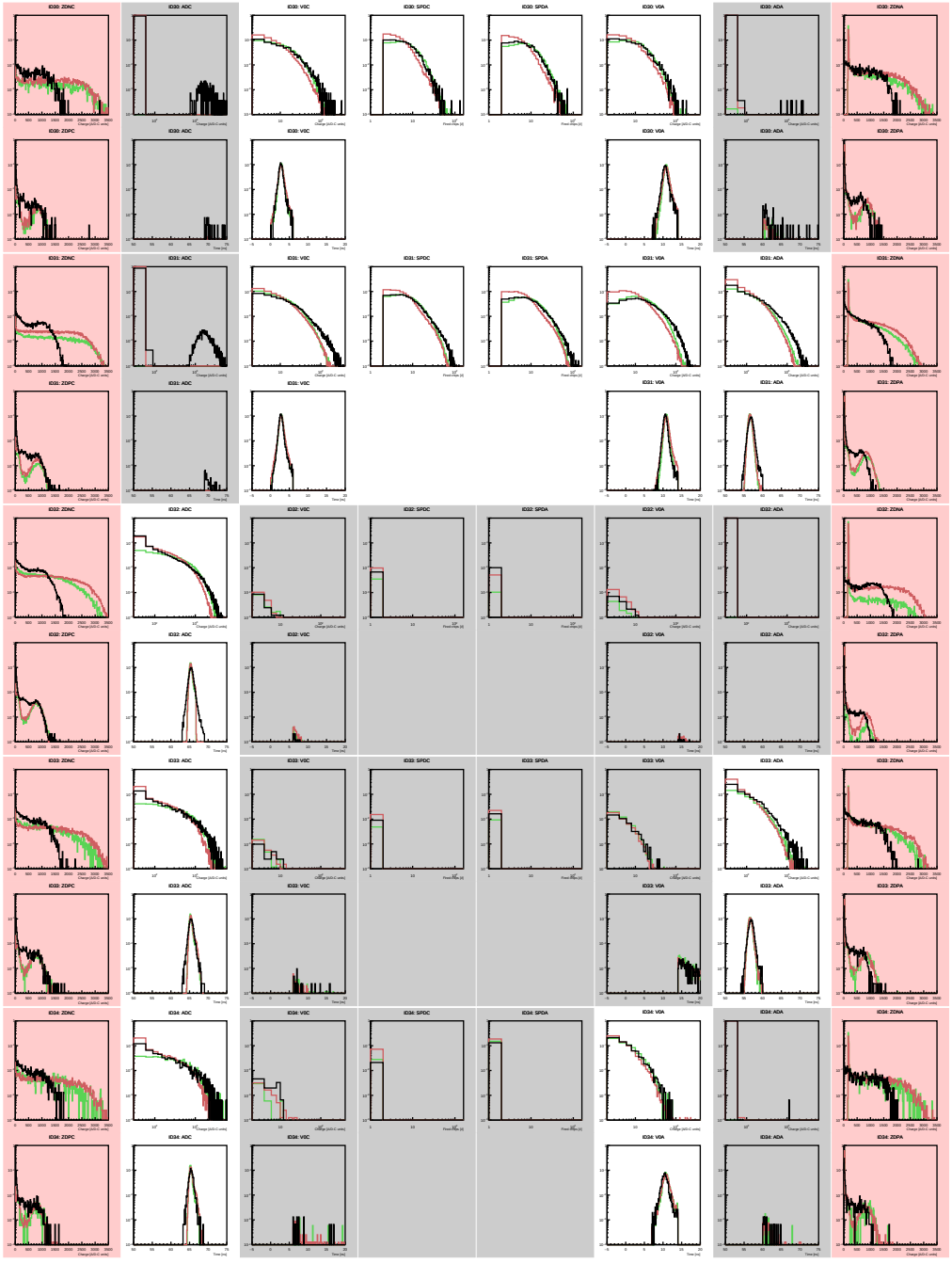

Figure A.7: ALICE **data** at $\sqrt{s} = 13$ TeV (black) and Pythia 6 (red), Phojet (green) simulations for combinations [30-34]: Detector level charge and time distributions.

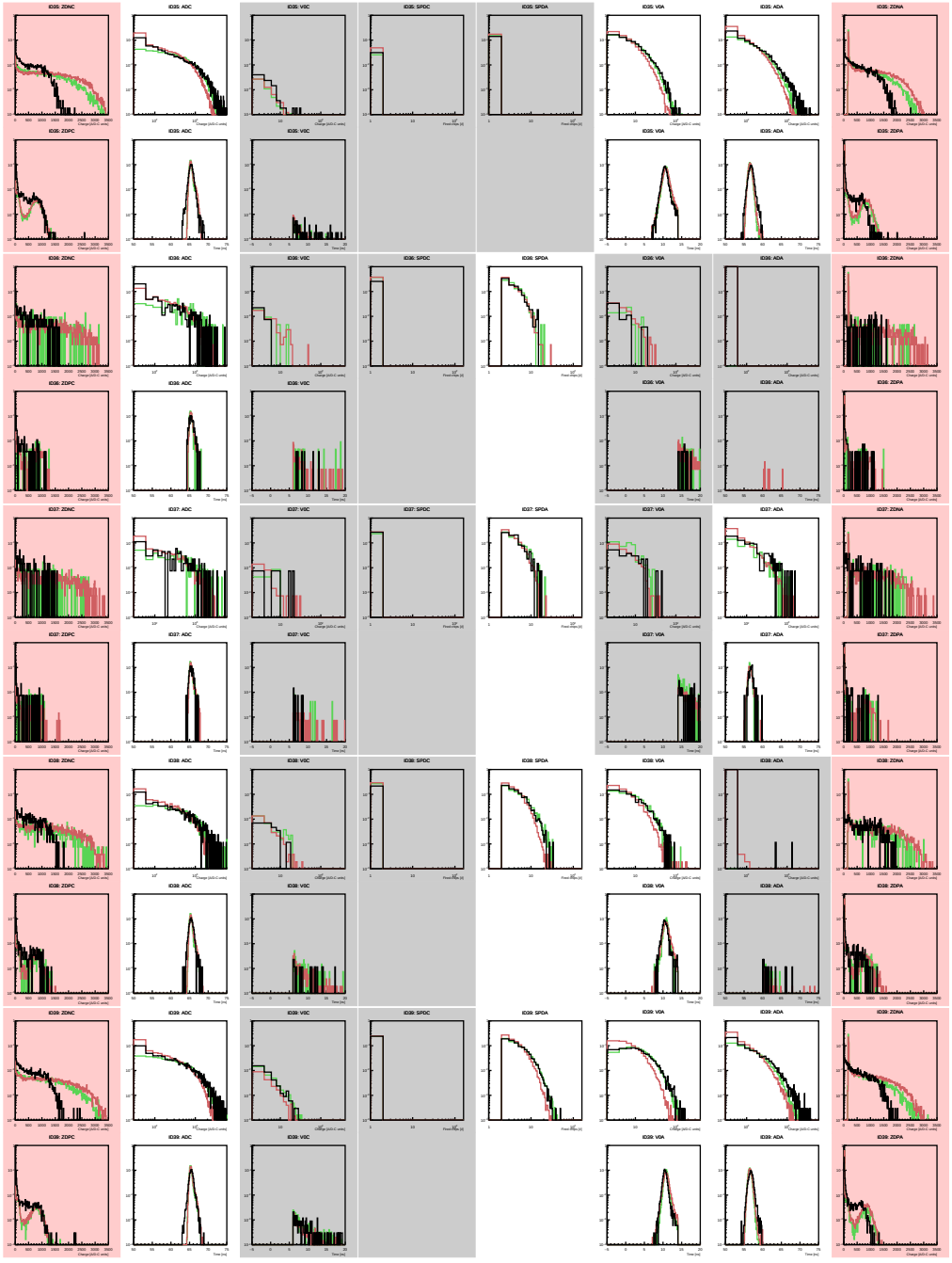

Figure A.8: ALICE **data** at $\sqrt{s} = 13$ TeV (black) and Pythia 6 (red), Phojet (green) simulations for combinations [35-39]: Detector level charge and time distributions.

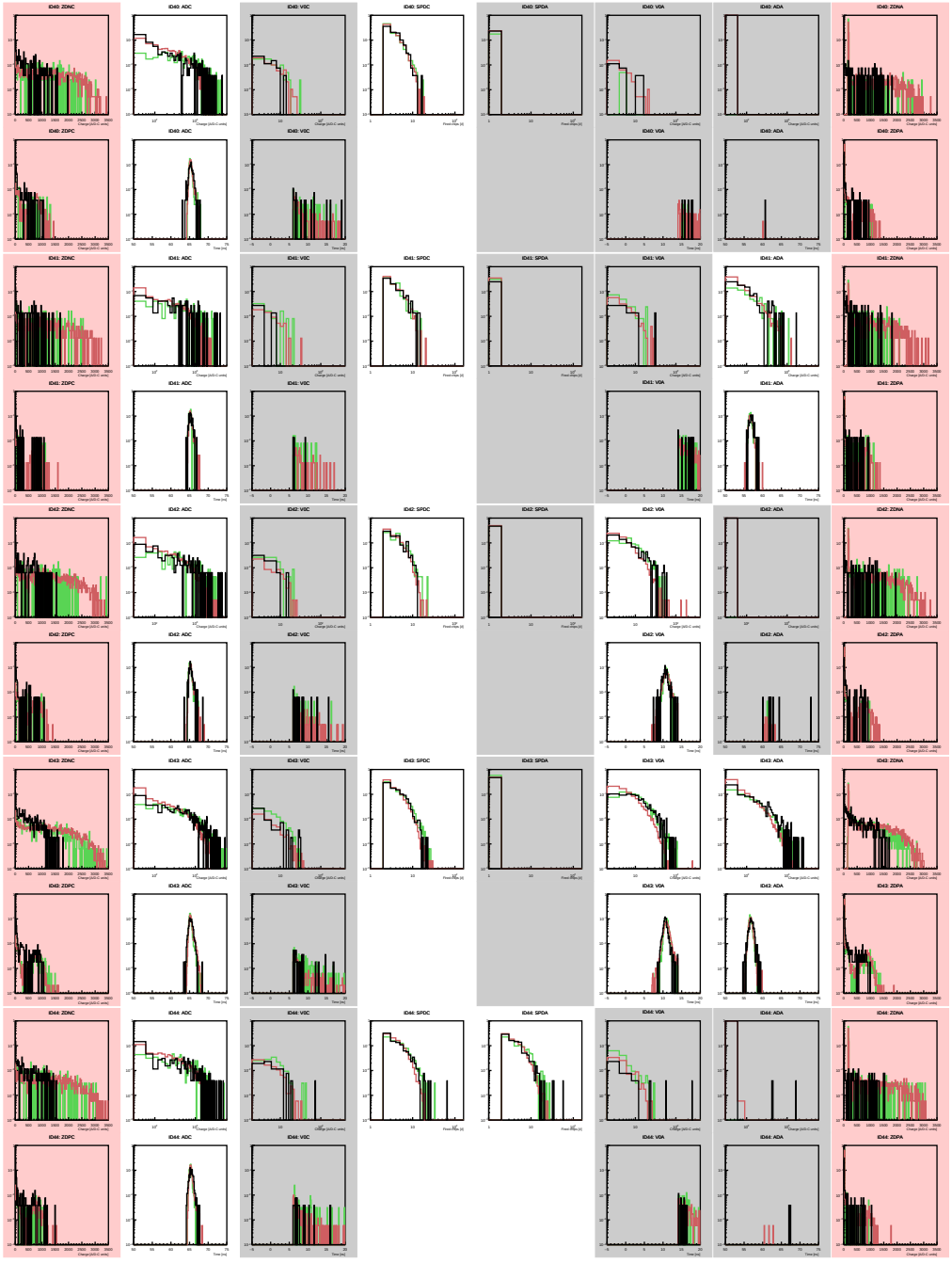

Figure A.9: ALICE **data** at $\sqrt{s} = 13$ TeV (black) and Pythia 6 (red), Phojet (green) simulations for combinations [40-44]: Detector level charge and time distributions.

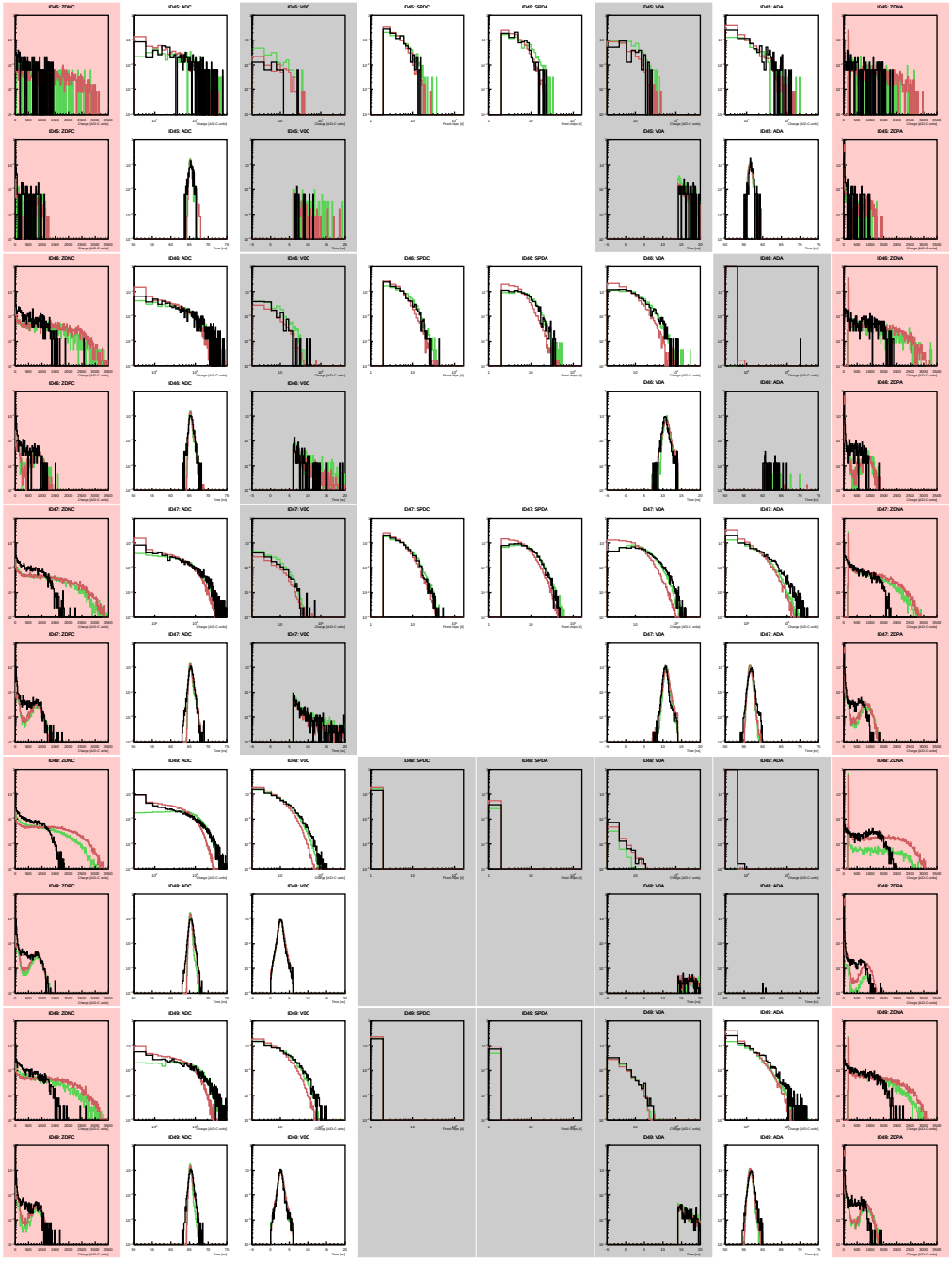

Figure A.10: ALICE **data** at $\sqrt{s} = 13$ TeV (black) and Pythia 6 (red), Phojet (green) simulations for combinations [45-49]: Detector level charge and time distributions.

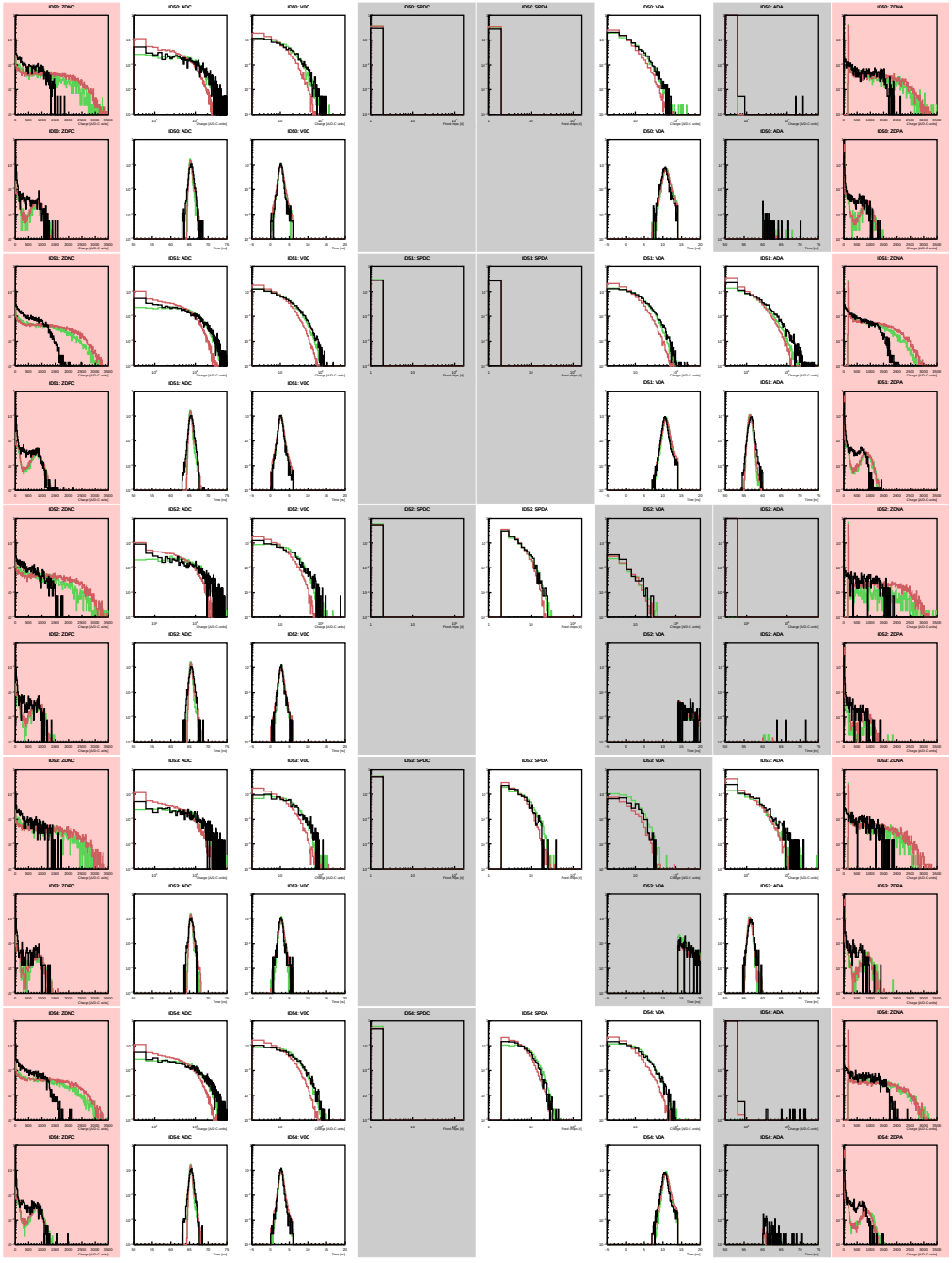

Figure A.11: ALICE **data** at $\sqrt{s} = 13$ TeV (black) and Pythia 6 (red), Phojet (green) simulations for combinations [50-54]: Detector level charge and time distributions.

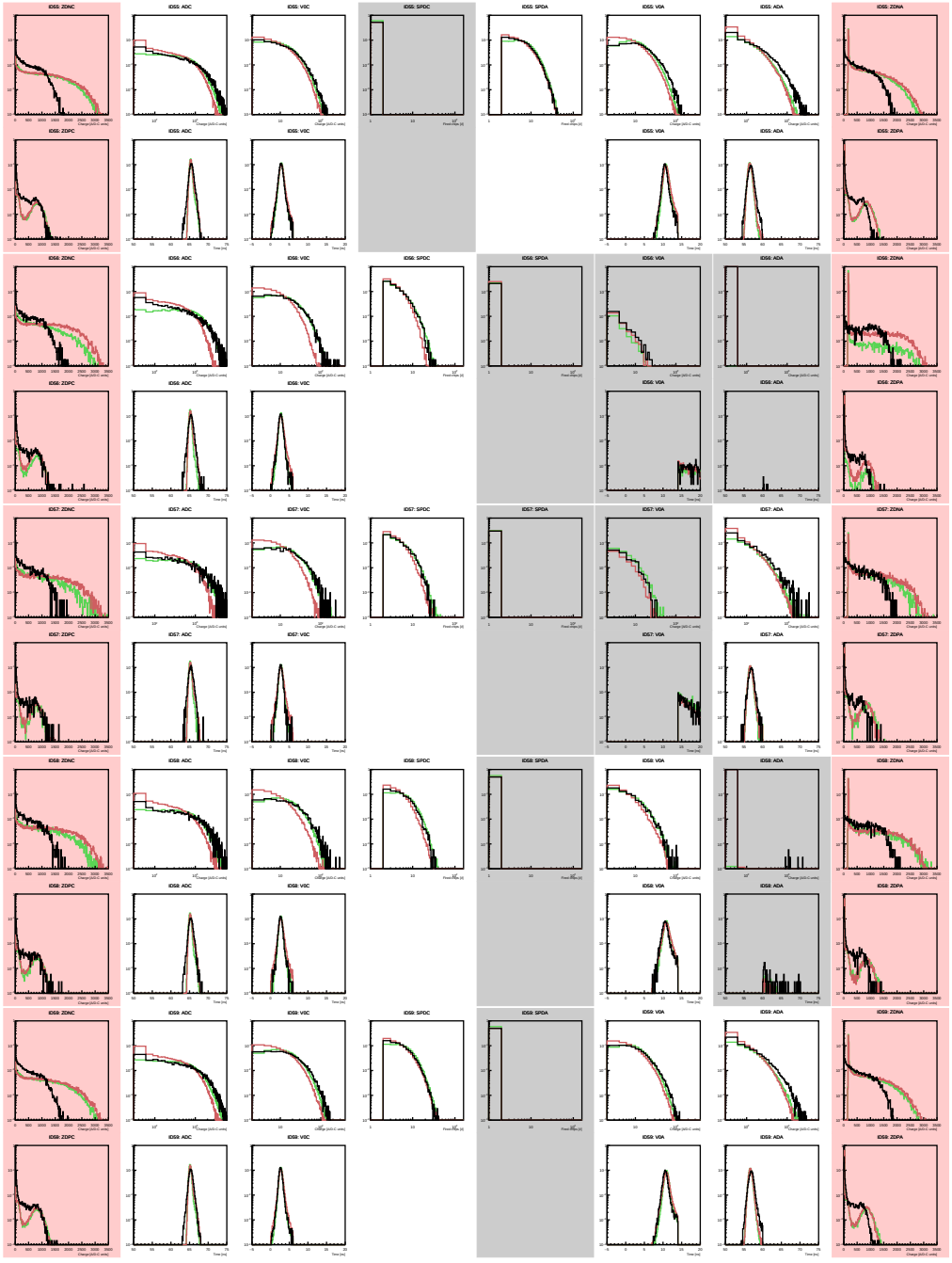

Figure A.12: ALICE **data** at $\sqrt{s} = 13$ TeV (black) and Pythia 6 (red), Phojet (green) simulations for combinations [55-59]: Detector level charge and time distributions.

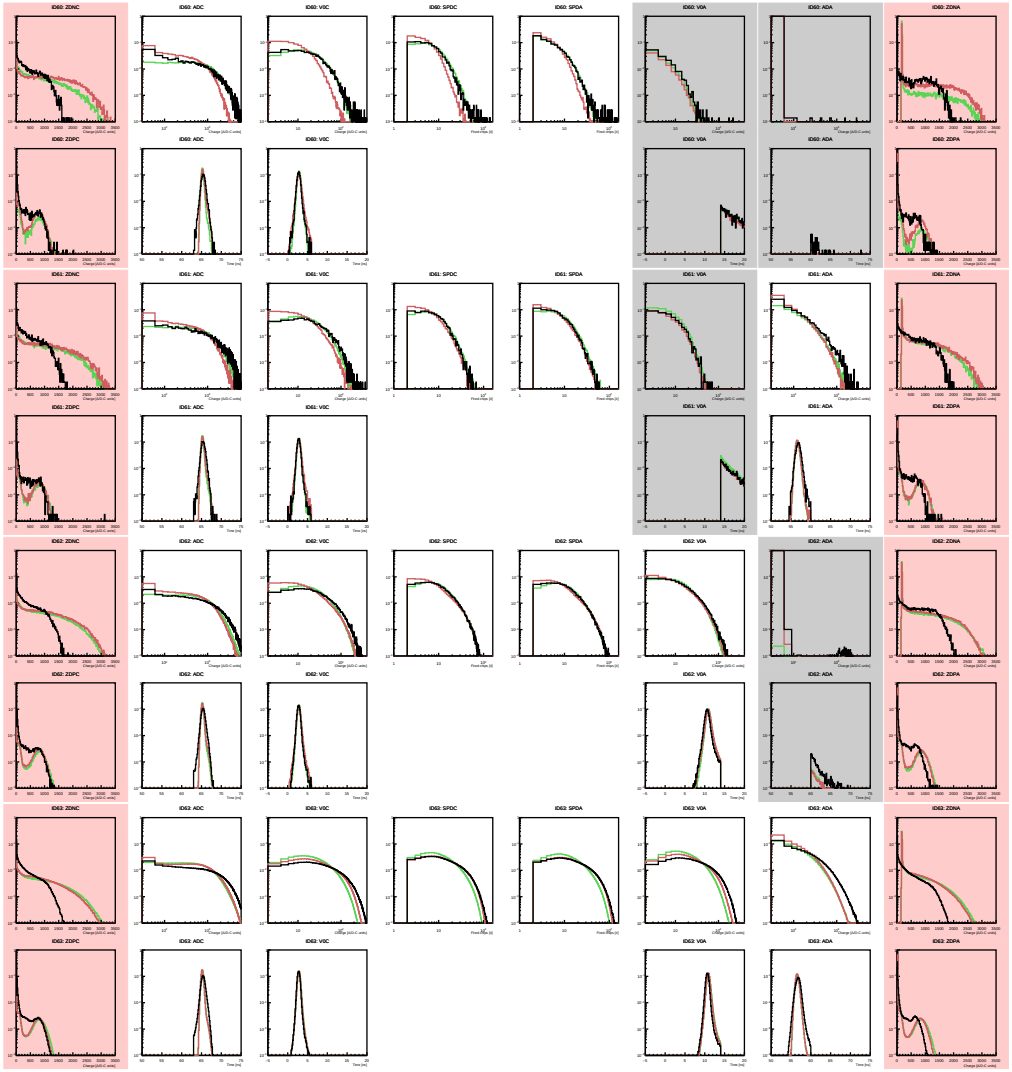

Figure A.13: ALICE **data** at $\sqrt{s} = 13$ TeV (black) and Pythia 6 (red), Phojet (green) simulations for combinations [60-63]: Detector level charge and time distributions.

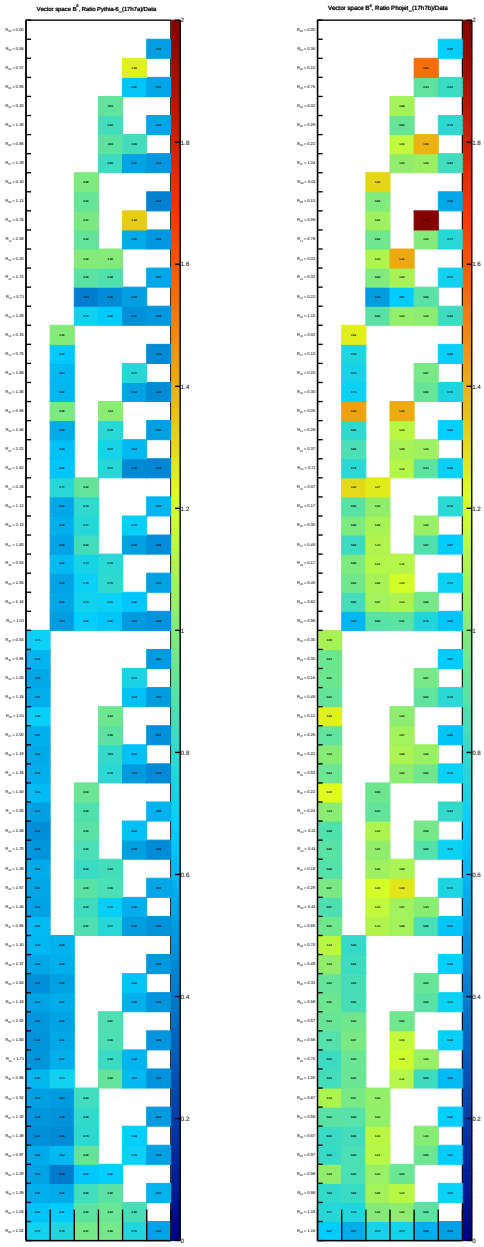

Figure A.14: ALICE data at $\sqrt{s} = 13$ TeV: Detector level visible cross section ratios for Pythia 6/Data (left) and Phojet/Data (right). Colored cells are mean signal ratios for: $\langle Q \rangle_{\mathrm{ADC}}, \langle Q \rangle_{\mathrm{V0C}}, \langle F \rangle_{\mathrm{SPDC}}, \langle F \rangle_{\mathrm{SPDA}}, \langle Q \rangle_{\mathrm{V0A}}, \langle Q \rangle_{\mathrm{ADA}}$, where Q is the charge and F is the number of fired chips.

A.2 Detector response simulations

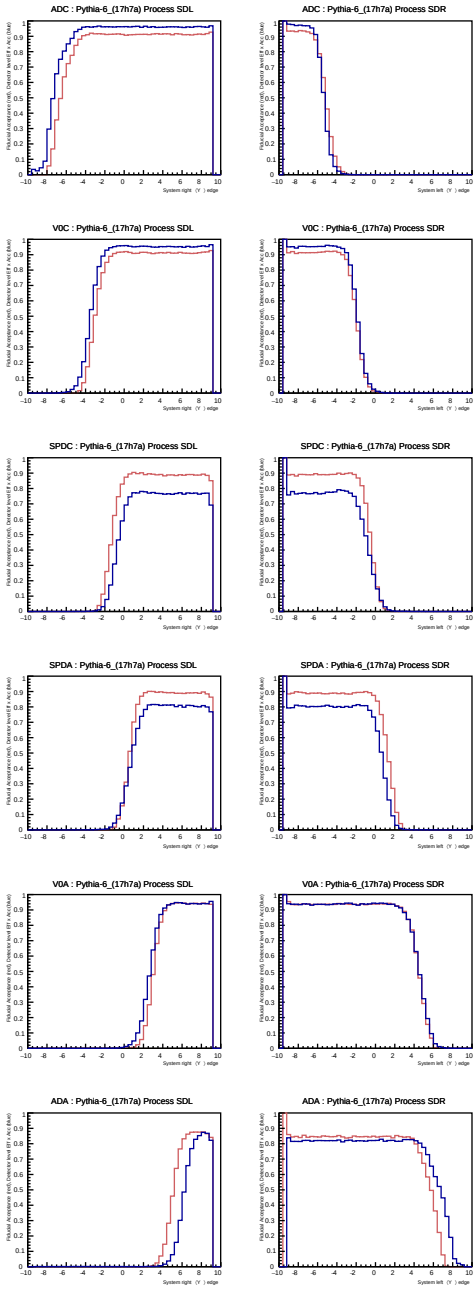

Figure A.15: ALICE simulation at $\sqrt{s} = 13$ TeV: Pythia 6 MC + GEANT based acceptance of diffractive systems (SDL left column, SDR right column) for different sub-detectors (as rows) in terms of the kinematic rapidity gap edge. Red is at the fiducial acceptance and blue at the detector efficiency $\times$ acceptance $\times$ smearing level.

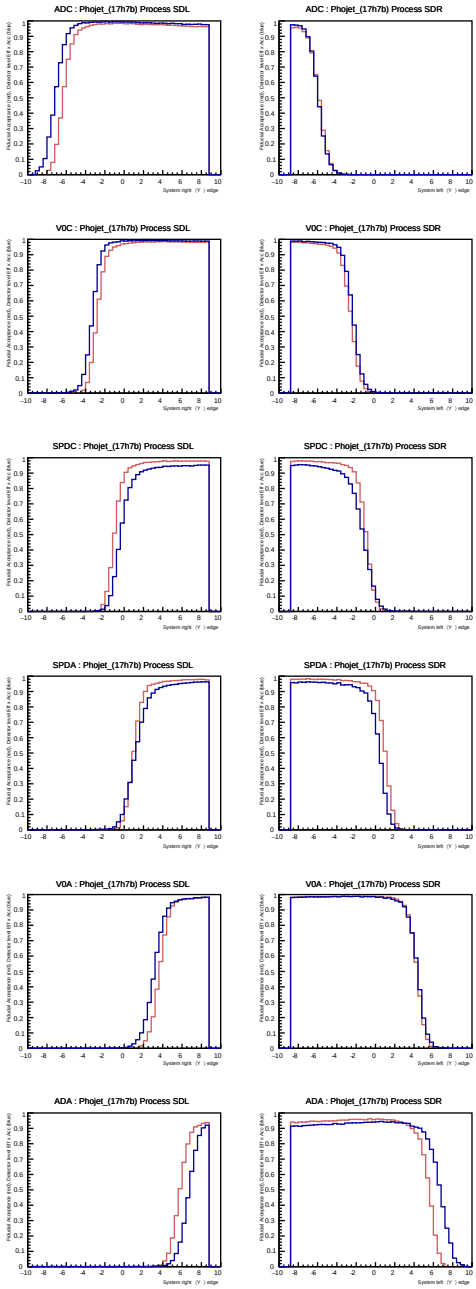

Figure A.16: ALICE simulation at $\sqrt{s} = 13$ TeV: Phojet MC + GEANT based acceptance of diffractive systems (SDL left column, SDR right column) for different sub-detectors (as rows) in terms of the kinematic rapidity gap edge. Red is at the fiducial acceptance and blue at the detector efficiency $\times$ acceptance $\times$ smearing level.

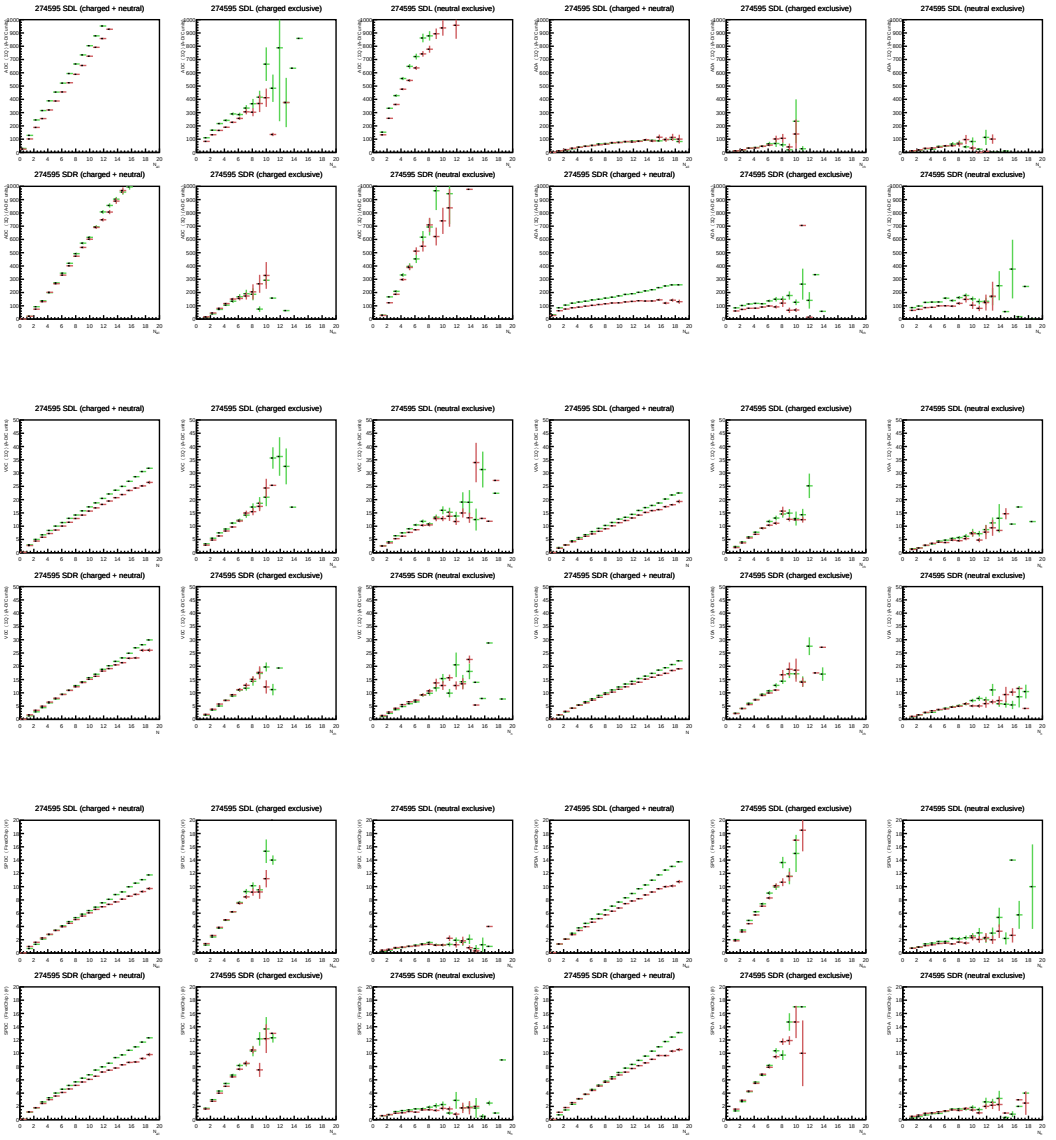

Figure A.17: ALICE simulation at $\sqrt{s} = 13$ TeV: Pythia (red) and Phojet (green) + GEANT simulated detector signal response as a function of the fiducial acceptance domain multiplicity containing charged, neutral or both type of particles for $\mathrm{AD}_{C,A}$ in the upper row, $\mathrm{VZERO}_{C,A}$ in the middle row and $\mathrm{SPD}_{C,A}$ in the bottom row.

A.3 LHC optics

The nominal LHC optics parameters at the ALICE IP2 are given in Table A.1, which show the larger β^*, transverse plane separation and crossing angle than other LHC experiments. These are used to reduce the luminosity by three (+two) orders of magnitude compared to the ATLAS and CMS (+LHCb) cases.

Plane of crossing	-	Vertical
Bunch population	N	1.25×10^{11}
Revolution frequency	f_{rev}	11245 Hz
Beam energy	E	6500 GeV
Normalized transverse emittance	ϵ	2.3×10^{-6} m
Beta*	β^*	10 m
Full crossing angle (xz)	α	480 μrad
Transverse plane separation	d	220 μm
Bunch length	σ_z	0.0787 m (1σ)
RF frequency	f_{RF}	400 MHz ($\Delta_t = 2.5$ ns)

Table A.1: LHC IP2 nominal run setup from, but here with an (ad-hoc) adjusted transverse plane separation (from 357 μm in lpc.web.cern.ch).

The transverse beam size is given by

$$\sigma_x = \sigma_y = [\epsilon\beta^*]^{1/2} = 57.61\ \mu\mathrm{m}. \tag{A.1}$$

The crossing angle affects the effective areas by

$$\Sigma_x = \left[2\sigma_x^2 \cos^2(\alpha/2) + 2\sigma_z^2 \sin^2(\alpha/2)\right]^{1/2} = 85.74\ \mu\mathrm{m} \tag{A.2}$$

$$\Sigma_y = \left[2\sigma_y^2\right]^{1/2} = 81.74\ \mu\mathrm{m}. \tag{A.3}$$

These values yield instantaneous luminosity per bunch pair

$$L_{bb} = \frac{f_{rev} N^2 \cos^2(\alpha/2) F}{2\pi \Sigma_x \Sigma_y} = 1.046 \times 10^{28}\ \mathrm{Hz/cm^2}, \tag{A.4}$$

where the effective separation factor is $F = \exp\left(-\frac{d^2}{2\Sigma_y^2}\right)$. The instantaneous luminosity integrated over all colliding bunch pairs n_{bb} is $L = n_{bb} L_{bb}$. The numbers here are from the LHC beam division luminosity calculator lpc.web.cern.ch.

A.4 Trigger statistics

The trigger acronyms are explained in Table A.2. The trigger system 'vetoes' or downtime can be due to the CTP (central trigger processor) being busy, the trigger cluster being busy or the trigger system having dead time for other countable reasons in the digital logic or due to a pre-determined trigger downscaling factor. The trigger system takes these into account with before (b) and after (a) counters. For more information about the ALICE trigger system, see [40].

Name	Meaning
ID	Trigger list identifier
BCMask	Bunch cross mask identifier (A,B,C,I,E)
DS	Deterministic down scaling factor (percent)
LMb(a)	Level minus-trigger before(after) trigger 'vetoes'
L0b(a)	Level 0-trigger before(after) trigger 'vetoes'
L1b(a)	Level 1-trigger before(after) trigger 'vetoes'
L2b(a)	Level 2-trigger before(after) trigger 'vetoes'

Table A.2: Trigger terminology.

ID	Name	BCMask	DS	Duration	LMb	LMa	L0b	L0a	L1b	L1a	L2b	L2a	LMb/Hz	LMa/Hz	L0b/Hz	L0a/Hz	L1b/Hz	L1a/Hz	L2b/Hz	L2a/Hz	LMa/LMb	L0a/L0b	L1a/L1b	L2a/L2b
34	CINT11-B-NOPF-CENTNOTRD	B		002431	46488908	46488908	741483	489433	489433	489433	489433	489433	224467	224467	358	236.3	236.3	236.3	236.3	236.3	1	0.66	1	1
35	CINT11-I-NOPF-CENTNOTRD	I		002431	4648819	4648819	68930	45843	45843	45843	45843	45843	22447	22447	33.3	22.1	22.1	22.1	22.1	22.1	1	0.67	1	1
36	CINT11-A-NOPF-CENTNOTRD	A	50	002431	8600 31532	8600 31532	127565	21729	21729	21729	21729	21729	415264	415264	60.2	10.5	10.5	10.5	10.5	10.5	1	0.17	1	1
37	CINT11-C-NOPF-CENTNOTRD	C	50	002431	83678740	83678740	92587	17574	17574	17574	17574	17574	404040	404040	44.7	8.5	8.5	8.5	8.5	8.5	1	0.19	1	1
38	CINT11-E-NOPF-CENTNOTRD	E		002431	46488192	46488192	1336	663	663	663	663	663	22447	22447	0.65	0.32	0.32	0.32	0.32	0.32	1	0.5	1	1

Table A.3: Trigger statistics for the run 274593.

ID	Name	BCMask	DS	Duration	LMb	LMa	L0b	L0a	L1b	L1a	L2b	L2a	LMb/Hz	LMa/Hz	L0b/Hz	L0a/Hz	L1b/Hz	L1a/Hz	L2b/Hz	L2a/Hz	LMa/LMb	L0a/L0b	L1a/L1b	L2a/L2b
34	CINT11-B-NOPF-CENTNOTRD	B		002558	34983232	34983232	686896	419822	419822	419822	419822	419822	224467	224467	440.7	269.4	269.4	269.4	269.4	269.4	1	0.61	1	1
35	CINT11-I-NOPF-CENTNOTRD	I		002558	3498 3253	3498 3253	62826	38450	38450	38450	38450	38450	22447	22447	40.3	24.7	24.7	24.7	24.7	24.7	1	0.61	1	1
36	CINT11-A-NOPF-CENTNOTRD	A	50	002558	647190172	647190172	1278 65	21336	21336	21336	21336	21336	415265	415265	82	13.7	13.7	13.7	13.7	13.7	1	0.17	1	1
37	CINT11-C-NOPF-CENTNOTRD	C	50	002558	629698541	629698541	102327	19308	19308	19308	19308	19308	404041	404041	65.7	12.4	12.4	12.4	12.4	12.4	1	0.19	1	1
38	CINT11-E-NOPF-CENTNOTRD	E		002558	34983253	34983253	2913	1557	1557	1557	1557	1557	22447	22447	1.9	1	1	1	1	1	1	0.53	1	1

Table A.4: Trigger statistics for the run 274594.

ID	Name	BCMask	DS	Duration	LMb	LMa	L0b	L0a	L1b	L1a	L2b	L2a	LMb/Hz	LMa/Hz	L0b/Hz	L0a/Hz	L1b/Hz	L1a/Hz	L2b/Hz	L2a/Hz	LMa/LMb	L0a/L0b	L1a/L1b	L2a/L2b
34	CINT11-B-NOPF-CENTNOTRD	B		002950	40188747	40188747	1623287	778911	778911	778911	778911	778911	224467	224467	90.7	435	435	435	435	435	1	0.48	1	1
35	CINT11-I-NOPF-CENTNOTRD	I		002950	4018887 4	4018887 4	145953	69786	69786	69786	69786	69786	22447	22447	81.5	39	39	39	39	39	1	0.48	1	1
36	CINT11-A-NOPF-CENTNOTRD	A	50	002950	743941 70	743941 70	116000	13856	13856	13856	13856	13856	415265	415265	64.8	7.7	7.7	7.7	7.7	7.7	1	0.12	1	1
37	CINT11-C-NOPF-CENTNOTRD	C	50	002950	723399732	723399732	86062	11244	11244	11244	11244	11244	404041	404041	48.1	6.3	6.3	6.3	6.3	6.3	1	0.33	1	1
38	CINT11-E-NOPF-CENTNOTRD	E		002950	4018887 4	4018887 4	1237	457	457	457	457	457	22447	22447	0.69	0.26	0.26	0.26	0.26	0.26	1	0.37	1	1

Table A.5: Trigger statistics for the run 274595.

B GRANIITTI and kinematics

B.1 Alternative models of pomeron

The original pomeron

The original 'sliding helicity' Gribov pomeron has been implemented in the code partially – this is only partial because the calculations involve unknown functions at the central vertex. However, we shall explore it a bit. In the following, we shall strip off all the form factors and concentrate only on the helicity dependent terms. The helicity amplitude is written as [111]

$$\begin{aligned} &T^{\lambda_1\lambda_3\lambda_2}_{\lambda_A\lambda_B}(s_1,s_2,t_1,t_2,\varphi) \\ &= \sum_{ab} g^a_{\lambda_A\lambda_1}(t_1)\Delta_a(s_1,t_1)g^{\lambda_3}_{ab}(t_1,t_2,\varphi)\Delta_b(s_2,t_2)g^b_{\lambda_B\lambda_2}(t_2), \end{aligned} \tag{B.1}$$

where the sum runs over exchanged Reggeon with propagators Δ_i as defined earlier, to first order at high energies, over only two pomerons. The particle-Reggeon-particle vertex helicity structure at small $-t$ is

$$g^a_{\lambda_A\lambda_1}(t) \simeq (-t)^{|\lambda_A-\lambda_1|/2}. \tag{B.2}$$

The spin structure of the crucial central vertex is [111]

$$g^{\lambda_3}_{ab}(t_1,t_2,\varphi) = \sum_{m_1,m_2=-\infty}^{\infty} e^{im_1\varphi}\gamma^{\lambda_3}_{m_1m_2}(t_1,t_2), \;\; \text{subject to} \;\; \lambda_3 = m_1 - m_2, \tag{B.3}$$

where

$$\gamma^{\lambda_3}_{m_1m_2}(t_1,t_2) \simeq (-t_1)^{|m_1|/2}(-t_2)^{|m_2|/2}, \tag{B.4}$$

a limit which holds for small values of $-t_1$ and $-t_2$. The projection of the angular momentum $j_{a(b)}$ of the Reggeon $a(b)$ is denoted with $m_{1(2)}$, which is an analytical continuation over all values over the sliding Regge trajectory, thus technically an infinite sum. However, the sum over $m_{1(2)}$ should be truncated to a first few, given that

$-t_{1(2)}$ are small and there is no full information about the central vertex available. We make remark that one can see easily that in order the parity conservation to hold for all spin-parities, the vertex Eq. B.4 cannot be always symmetric e.g. under $(m_1, m_2) \leftrightarrow (-m_1, -m_2)$, but should change sign – thus this equation is a slightly formal one. Now we have hopefully emphasized enough that the original pomeron has no fixed spin structure but is an infinite sum. It is essentially the spin-parity of the central state which 'forces' the pomeron Lorentz structure to look like a non-conserved or conserved vector (or tensor) current, from a Regge theory point of view, as discussed in [111].

Vector current pomeron

This is a model from [234], which illustrates many points and is described here shortly for completeness. In some sense the Tensor pomeron model is a superset of this with complete Feynman rules and higher rank spin structure. The spin density matrix of the i-th pomeron in this conserved vector current model is obtained as

$$\rho_i^{\lambda\lambda'} = (-1)^{\lambda+\lambda'} \epsilon^{\mu}(q_i, \lambda)\epsilon^{\nu}(q_i, \lambda')\rho^{\mu\nu}, \tag{B.5}$$

where $\lambda = 0, \pm 1$ and the space-like $q^2 < 0$ spin-1 polarization vectors are denoted with ϵ. On the right hand side, the covariant density matrix is hermitian $\rho^{\mu\nu *} = \rho^{\nu\mu}$ and the spin-1 polarization vectors obey

$$\epsilon^{\mu *}(\pm 1) = -\epsilon^{\mu}(\mp 1) \tag{B.6}$$

$$\epsilon^{\mu *}(0) = -\epsilon^{\mu}(0). \tag{B.7}$$

We denote the 4 independent elements of the density matrix by ρ^{++}, ρ^{00}, which are on the diagonal, and the off-diagonal terms by $|\rho^{+0}|e^{i\tilde{\varphi}_i}$ and $|\rho^{+-}|e^{i\tilde{\varphi}_i}$, where $\tilde{\varphi}_i$ is the forward proton azimuthal angle in the pomeron-pomeron CM frame. This angle is not exactly the same, clearly, as the forward angle φ in the proton-proton CM frame or LHC lab frame.

The unnormalized covariant density matrix for the pomeron emission from the i-th proton leg is

$$\rho_i^{\mu\nu} = -\frac{1}{q_i^2} \sum_{\text{helicities}} J_\mu J_\nu^*, \tag{B.8}$$

where J_μ is a Dirac like current of the proton-pomeron-proton vertex as the one in QED given by Eq. 5.48. However, pomeron may couple with different strengths than photon to the electric and magnetic terms of the proton form factors. The differential

cross section is then obtained by coupling $\rho_1^{\lambda_1\lambda_1'}$ and $\rho_2^{\lambda_2\lambda_2'}$ elements together at the central vertex [234]

$$\begin{aligned} d\sigma \sim\ & 2\rho_1^{++}\rho_2^{++} \times [W(++,++) + W(+-,+-)] \\ & + 2\rho_1^{++}\rho_2^{00} W(+0,+0) + 2\rho_1^{00}\rho_2^{++} W(0+,0+) \\ & + \rho_1^{00}\rho_2^{00} W(00,00) \\ & + 2|\rho_1^{+-}\rho_2^{+-}| W(++,--)\cos 2\tilde{\varphi} \\ & - 4|\rho_1^{+0}\rho_2^{+0}| \times [W(++,00) + W(0+,-0)]\cos\tilde{\varphi}, \end{aligned} \tag{B.9}$$

which depends on eight helicity structure functions $W(\lambda_1\lambda_2, \lambda_1'\lambda_2')$, where the helicity conservation requires $\lambda_1 - \lambda_2 = J_z = \lambda_1' - \lambda_2'$, otherwise $W = 0$. The azimuthal angle is $\tilde{\varphi} = \tilde{\varphi}_1 + \tilde{\varphi}_2$. Six of these structure function can be measured with unpolarized initial state protons. Now, let us write this in terms of the helicity amplitudes

$$W(\lambda_1\lambda_2, \lambda_1'\lambda_2') \sim A_{\lambda_1\lambda_2}(t_1,t_2) A^*_{\lambda_1'\lambda_2'}(t_1,t_2)\delta((q_1+q_2)^2 - M^2). \tag{B.10}$$

Parity conservation requires that $A_{-\lambda_1-\lambda_2} = \eta_1\eta_2\eta_M A_{\lambda_1\lambda_2}$, where $\eta_{1,2}$ denote the naturality of pomerons and η_M the naturality of the boson, which is $+1$ if $P = (-1)^J$ and -1 if $P = (-1)^{J-1}$. Bose-Einstein symmetry (statistics) requires $A_{\lambda_1\lambda_2}(t_1,t_2) = (-1)^J A_{\lambda_2\lambda_1}(t_2,t_1)$, and time invariance requires invariance under under $\lambda_1\lambda_2 \leftrightarrow \lambda_1'\lambda_2'$, which gives somewhat more complicated relations [234]. This model has some definite properties regarding the observables. Most of them are direct consequences of the parity conservation and vector currents.

B.2 Kinematics of $2 \to 3$

Lorentz invariants

Let us have the $2 \to 3$ process

$$p_A + p_B \to p_1 + P_3 + p_2, \tag{B.11}$$

where p_1 and p_2 are the forward systems. Kinematically this is fully characterized by 5 linearly independent Lorentz scalars. The most typical set is $s = (p_A + p_B)^2 = (p_1 + P_3 + p_2)^2$, $t_1 = (p_A - p_1)^2$, $t_2 = (p_B - p_2)^2$, $s_1 = (p_1 + P_3)^2$, $s_2 = (p_2 + P_3)^2$.

Colliding beam frame

Now, we need (at least) 5 variables. For practical reasons, we will use 6. The redundancy is between forward system transverse angles φ_1 and φ_2 which we can remove by the rotational invariance and use only the difference $\Delta\varphi \equiv \varphi_1 - \varphi_2$, whenever necessary. We define 6 variables, which fully characterize the exact kinematics and separate between the longitudinal and transverse degrees of freedom

$$\xi_{1(2)} \equiv 1 - \frac{p_{z,1(2)}}{p_{z,A(B)}} \in [0, 1] \tag{B.12}$$

$$p_{t,1(2)} \equiv \left(p_{x,1(2)}^2 + p_{y,1(2)}^2\right)^{1/2} \in \mathbb{R}_+ \tag{B.13}$$

$$\varphi_{1(2)} \equiv \tan^{-1}\left(\frac{p_{y,1(2)}}{p_{x,1(2)}}\right) \in (-\pi, \pi]. \tag{B.14}$$

Due to the 4-momentum conservation, not all arbitrary combinations of the values of these variables are physically valid.

4-momentum components

The momentum $p^\mu = [E, \vec{p}]$ of forward systems are now given directly by

$$p_{1(2)}^\mu = [\left(M_{1(2)}^2 + p_{t,1(2)}^2 + (-m_p^2 + \frac{s}{4})(\xi_{1(2)} - 1)^2\right)^{1/2},$$
$$p_{t,1(2)}\cos(\varphi_{1(2)}),\, p_{t,1(2)}\sin(\varphi_{1(2)}),\, \mp\frac{(\xi_{1(2)} - 1)(-4m_p^2 + s)^{1/2}}{2}]. \tag{B.15}$$

We allow also excitation of forward protons with masses M_1 and M_2. For the elastic forward protons, we set $M_{1(2)} \equiv m_p$. The central system 4-momentum is

$$\begin{aligned} &P_3^\mu = q_1^\mu + q_2^\mu = \\ &[s^{1/2} - \left(M_1^2 + p_{t,1}^2 + (-m_p^2 + \frac{s}{4})(\xi_1 - 1)^2\right)^{1/2} \\ &- \left(M_2^2 + p_{t,2}^2 + (-m_p^2 + \frac{s}{4})(\xi_2 - 1)^2\right)^{1/2}, \\ &- p_{t,1}\cos(\varphi_1) - p_{t,2}\cos(\varphi_2),\ -p_{t,1}\sin(\varphi_1) - p_{t,2}\sin(\varphi_2),\ \frac{(\xi_1 - \xi_2)(-4m_p^2 + s)^{1/2}}{2}]. \end{aligned} \tag{B.16}$$

4-momentum transfer squared

The Lorentz scalars $t_{1(2)} < 0$, which are the 4-momentum transfer squared, are written as

$$\begin{aligned} t_{1(2)} = &\frac{\left(\left((\xi_{1(2)} - 1)^2(-4m_p^2 + s) + 4M_{1(2)}^2 + 4p_{t,1(2)}^2\right)^{1/2} - s^{1/2}\right)^2}{4} \\ &- \frac{\xi_{1(2)}^2(-4m_p^2 + s)}{4} - p_{t,1(2)}^2. \end{aligned} \tag{B.17}$$

In the limit $\xi_1, \xi_2 \to 0$ and $s \to \infty$, we get the familiar

$$\boxed{t_{1(2)} \simeq -p_{t,1(2)}^2.} \tag{B.18}$$

This approximation is very good. When $p_{t,1(2)}^2 \lesssim 1$ GeV^2, the relative error we make is proportional to the scale of ξ and for low mass CEP at the LHC, this scale is $\xi \sim 10^{-4}$.

Central system rapidity

The boost (rapidity) definition $Y_X = \frac{1}{2}\ln\left(\frac{E+p_z}{E-p_z}\right)$ along the beam line gives

$$
\begin{aligned}
Y_X = \frac{1}{2}\ln[&\Big((M_1^2 + p_{t,1}^2 + (-m_p^2 + \frac{s}{4})(\xi_1 - 1)^2\Big)^{1/2} + \\
&\Big(M_2^2 + p_{t,2}^2 + (-m_p^2 + \frac{s}{4})(\xi_2 - 1)^2\Big)^{1/2} - \frac{(\xi_1 - \xi_2)(-4m_p^2 + s)^{1/2}}{2} - s^{1/2}] \\
/[&\Big(M_1^2 + p_{t,1}^2 + (-m_p^2 + \frac{s}{4})(\xi_1 - 1)^2\Big)^{1/2} + \\
&\Big(M_2^2 + p_{t,2}^2 + (-m_p^2 + \frac{s}{4})(\xi_2 - 1)^2\Big)^{1/2} + \frac{(\xi_1 - \xi_2)(-4m_p^2 + s)^{1/2}}{2} - s^{1/2}],
\end{aligned}
$$

where taking massless limits and collinear $p_{t,1(2)} \to 0$ gives

$$
\boxed{Y_X \simeq \frac{1}{2}\ln\left(\frac{\xi_1}{\xi_2}\right).} \tag{B.19}
$$

This is all around a very good approximation, relative error $10^{-3} \ldots 10^{-6}$.

Subsystem energy invariants

The scalars $s_1 = (p_1 + P_3)^2 > 0$ and $s_2 = (p_2 + P_3)^2 > 0$ are

$$
s_1 = s - 2s^{1/2}\Big(M_2^2 + p_{t,2}^2 + (-m_p^2 + \frac{s}{4})(\xi_2 - 1)^2\Big)^{1/2} + M_2^2 \tag{B.20}
$$

$$
s_2 = s - 2s^{1/2}\Big(M_1^2 + p_{t,1}^2 + (-m_p^2 + \frac{s}{4})(\xi_1 - 1)^2\Big)^{1/2} + M_1^2. \tag{B.21}
$$

The obvious massless and $p_{t,1(2)} \to 0$ limits of these are

$$
\boxed{s_1 \simeq s\xi_2, \quad s_2 \simeq s\xi_1.} \tag{B.22}
$$

Relations above are sometimes also called as 'Regge domain criteria', equivalent with $s \gg |t_{1(2)}|, s \to \infty$. Note how terms are crossed $1 \leftrightarrow 2$, due to the momentum flow. The approximations are very good for high ξ values and even at $\xi \sim 10^{-6}$, the relative error is order of 10^{-3}. In terms of the central system invariant mass and rapidity, we can also write

$$
\boxed{s_1 \simeq M_X s^{1/2}\exp(-Y_X), \quad s_2 \simeq M_X s^{1/2}\exp(Y_X),} \tag{B.23}
$$

often used in the case of photoproduction. These can be derived using using the collinear relations

$$\boxed{\xi_1 \simeq \frac{M_X}{s^{1/2}}\exp(-Y_X), \quad \xi_2 \simeq \frac{M_X}{s^{1/2}}\exp(Y_X)} \tag{B.24}$$

in the limit $p_{t,1(2)} \to 0$.

Central system invariant mass

The central system invariant mass squared $M_X^2 \equiv P_3^2 = (q_1+q_2)^2$, is exactly written as

$$\begin{aligned} M_X^2 =& [\Big(M_1^2 + p_{t,1}^2 + (-m_p^2 + \frac{s}{4})(\xi_1 - 1)^2\Big)^{1/2} \\ &+ \Big(M_2^2 + p_{t,2}^2 + (-m_p^2 + \frac{s}{4})(\xi_2 - 1)^2\Big)^{1/2} - s^{1/2}]^2 \\ &- \left(p_{t,1}\cos(\varphi_1) + p_{t,2}\cos(\varphi_2)\right)^2 - \left(p_{t,1}\sin(\varphi_1) + p_{t,2}\sin(\varphi_2)\right)^2 \\ &- \frac{(\xi_1 - \xi_2)^2(-4m_p^2 + s)}{4}. \end{aligned} \tag{B.25}$$

Now if we take limit $p_{t,1(2)} \to 0$ and also $m_p, M_1, M_2 \to 0$, that is, $s \gg m_p^2, M_1^2, M_2^2$, we get

$$\boxed{M_X^2 \simeq \xi_1\xi_2 s.} \tag{B.26}$$

This approximation is *not* especially good for the low mass CEP, because the $p_{t,1,2}$ scale is close to the central system mass scale. The relative error can be order of one ($\sim 100\,\%$). Note that the complement scalar is

$$s_{12} = (p_1 + p_2)^2 \simeq (\xi_1 - 1)(\xi_2 - 1)s, \tag{B.27}$$

which is not in common use.

Spanning set of variables

We show here a generic method for checking if the constructed set of Lorentz scalars is a spanning set. In a row order s, t_1, t_2, s_1, s_2, we construct a matrix

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}, \tag{B.28}$$

where each row/column correspond to the corresponding 4-momenta indices with order p_A, p_B, p_1, p_2, P_3 and the matrix element is $A_{ij} = \pm$ when $(p_i \pm p_j)^2$. This matrix should have full rank $\Leftrightarrow$ a non-zero determinant. Computer algebra gives full rank$(A) = 5$ and determinant -2, which fine.

How about the 4-momentum representations? For this, we can construct the Gram matrix between all 4-dot products $G_{ij} = \langle p_i, p_j \rangle$. The determinant of this matrix is identically zero, because we have 5 vectors in 4-dimensional space. Thus, the check goes for example by choosing (only) the final states (p_3, p_4, p_5), and calculating the 3×3 Gram matrix, and its determinant. The determinant obtained is an extremely lengthy expression, and gives zero for *linearly dependent final state configurations*, which is fine. An example of a linearly dependent case: $(\xi_1 \equiv \xi_2, p_{t,1} \equiv p_{t,2}, \Delta\varphi \equiv \varphi_1 - \varphi_2 \equiv \pi) \rightarrow$ system produced at rest gives $\det(G) = 0$.

B.3 The slope parameter

Experimentally in soft processes $d\sigma/dt \sim \exp(-b|t|)$ holds for small values of $|t|$. For example, in photoproduction the pomeron exchange t-distribution slope is often written as

$$b = b_0 + 2\alpha' \ln\left(\frac{W^2}{W_0^2}\right), \tag{B.29}$$

where b_0 is a process (e.g. vector meson mass) dependent constant, W^2 is the $\gamma^* p$ subsystem invariant such as s_1 or s_2 here and W_0^2 is the (fit) normalization scale squared, often $W_0 \simeq 90$ GeV in HERA fits. The second term with the pomeron slope $\alpha' \simeq 0.05 \ldots 0.25$ GeV^{-2} originates from the Regge phenomenology, where as the first term is simply based on an exponential ansatz. That is, this exponential ansatz may encapsulate both the proton form factor and the vector meson form factor or transverse size, however, a model dependent factorization ansatz between them is also possible.

The photon side t-distribution is different and is driven by QED photon singularity $1/q^2$ and proton EM-form factors. The slope parameter has dimensions (GeV^{-2}) which is the same as the cross section and may be interpreted as the average size of the transverse interaction region, which naively could do logarithmic grow infinitely, perhaps without confinement. So if α' is the 'inverse string tension' coupled with the logarithmically growing Gribov diffusion term, what is the constant b_0 term then? Perhaps it is the Fermi $\langle k_t^2 \rangle \sim 1/b$ of the 'vacuum noise', by definition.

Slope inference fit

We shall make a remark that an exponential $-t \simeq p_t^2$-distribution results in a Rayleigh distributed p_t-distribution, which then means that p_x and p_y components are following a Gaussian stochastic process with a variance σ^2. The relation between the exponential and the Rayleigh can be shown directly by the change of a variable theorem

$$f(p_t^2) = b\exp(-bp_t^2) \;\Leftrightarrow\; g(p_t) = \left|\frac{dp_t^2}{dp_t}\right| f(p_t^2) = 2p_t\, b\exp(-bp_t^2). \tag{B.30}$$

Using this, the relation between the parameter b and the forward system average transverse momentum $\langle p_t \rangle$ with the used distribution assumptions is

$$b = \frac{\pi}{4\langle p_t \rangle^2} = \frac{1}{2\sigma^2}. \tag{B.31}$$

We note that the square here is of the average, not the other way around.

How to obtain an estimate of b without forward proton measurements? First assume that the forward proton separation $\Delta\varphi$ follows a specific distribution, such as the uniform, generate forward proton p_t^2 values according to the chosen b, construct the forward proton transverse momentum vectors $\vec{p}_{t,1(2)}$, take the sum $|-\vec{P}_t| = |\vec{p}_{t,1} + \vec{p}_{t,2}|$ and compare this sample with the measured central system transverse momentum $|\vec{P}_t|$-distribution in a fit loop. Extension of this is to take into account the forward proton dissociation, which results qualitatively in a different distribution presumably with a hard (point like) power law tail, perhaps compatible with a Lévy flight like stochastic process in the transverse plane. Clearly, one needs to take into account in the fit the mixture of elastic-elastic, elastic-inelastic, inelastic-inelastic events. The most extensive fits are most easily done by generating samples using the full Graniitti machinery.

B.4 Harmonic acceptance decompositions

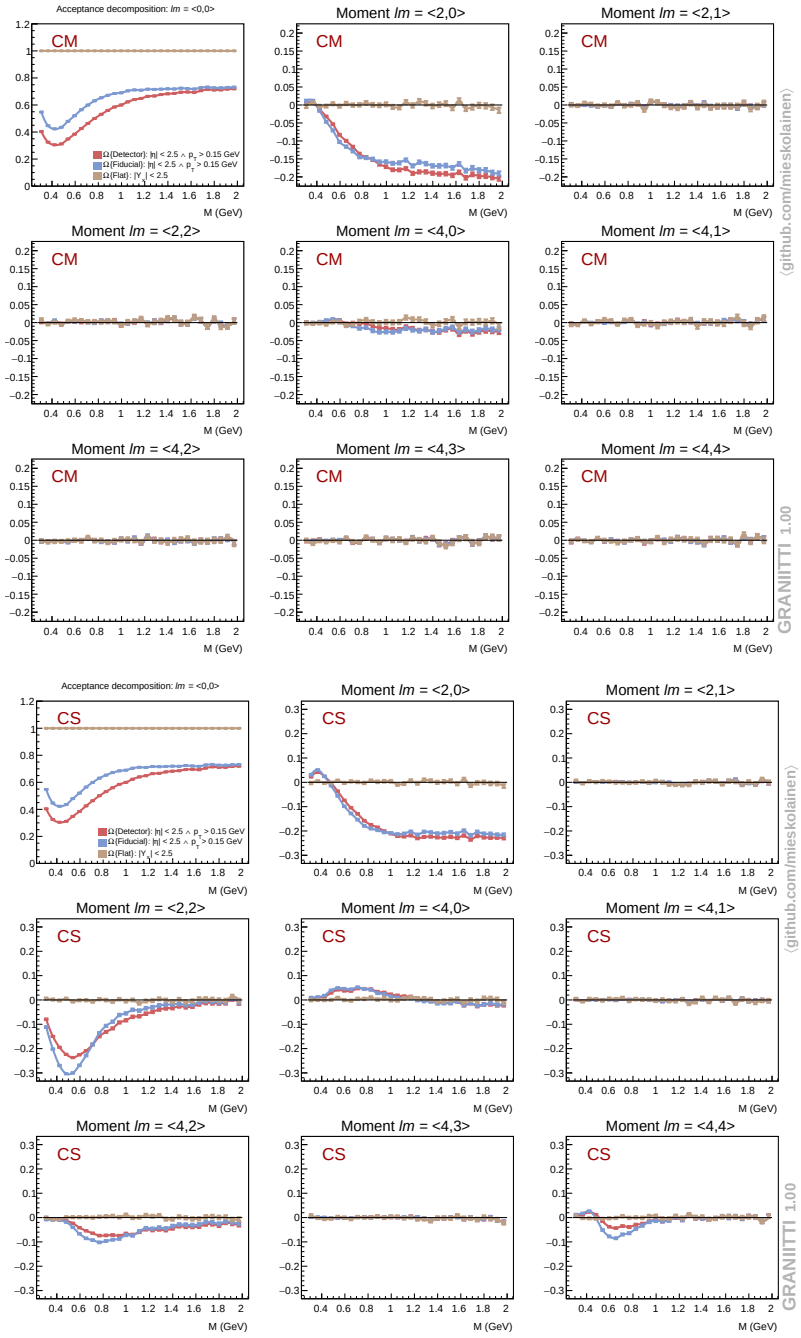

Figure B.1: CM frame (up) and CS frame (down): Acceptance decomposition.

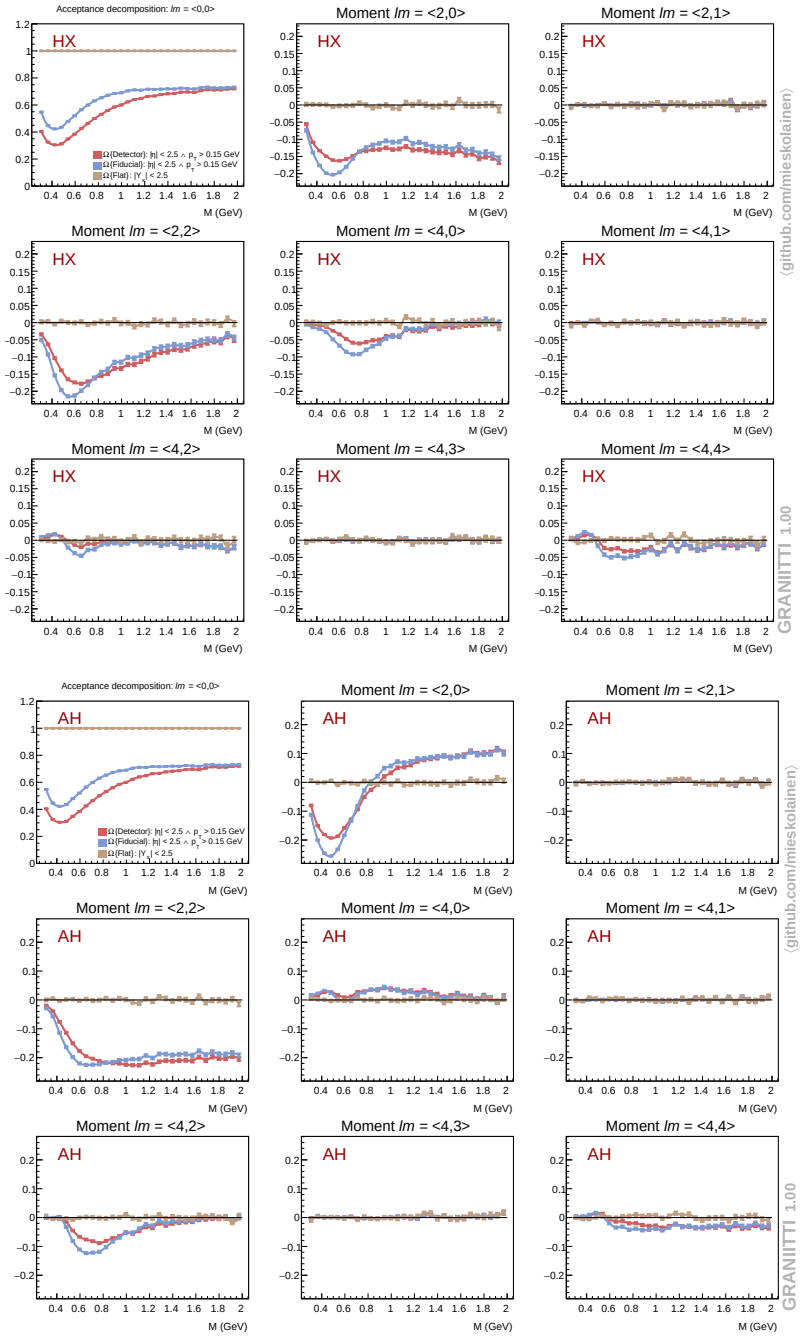

Figure B.2: HX frame (up) and AH frame (down): Acceptance decomposition.

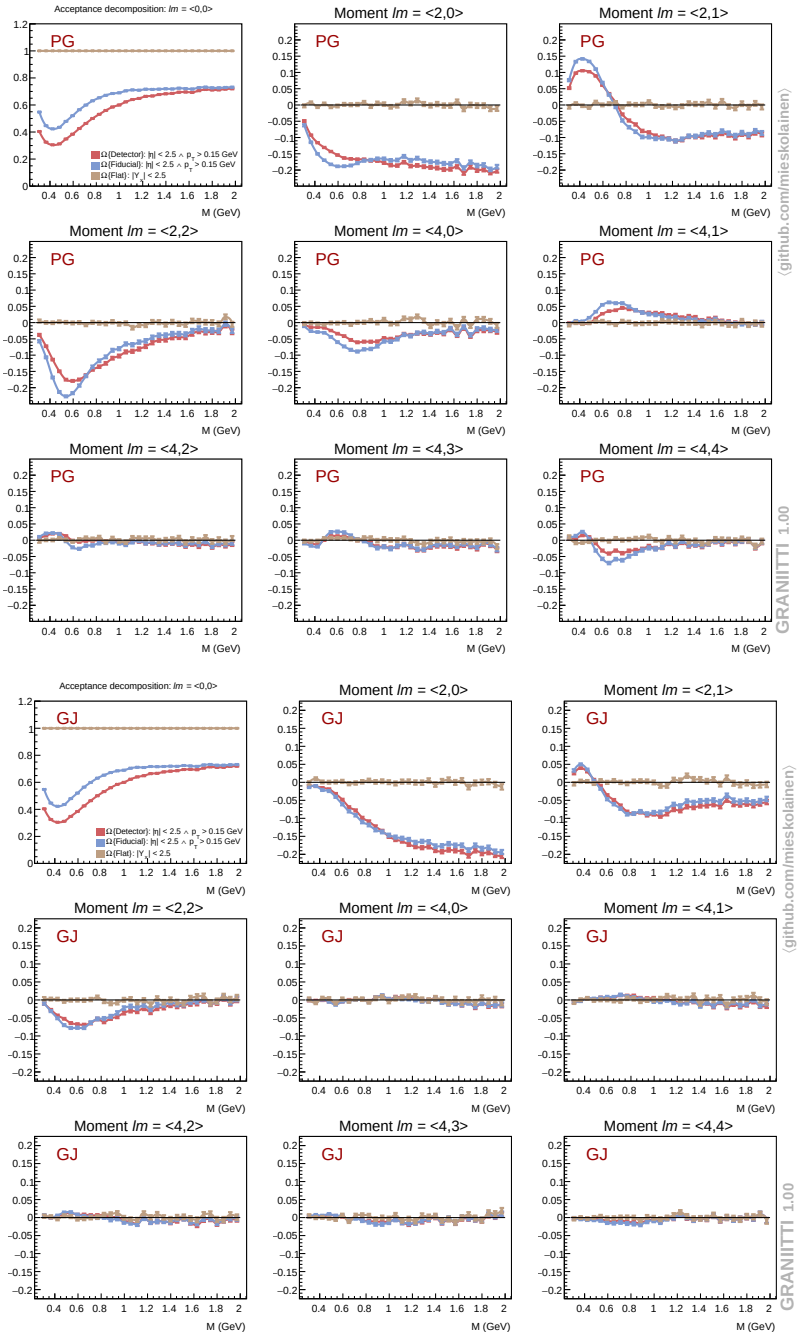

Figure B.3: PG frame (up) and GJ frame (down): Acceptance decomposition.

C Combinatorics and pileup

C.1 Vector space subspaces over finite fields

The q-binomial coefficient counts the number of subspaces of dimension r in a vector space of dimension N, when the vector space is over a finite field with a prime power q, is

$$\begin{bmatrix} N \\ r \end{bmatrix}_q = \begin{cases} \frac{(1-q^N)(1-q^{N-1})\cdots(1-q^{N-r+1})}{(1-q)(1-q^2)\cdots(1-q^r)}, & \text{if } r \leq N. \\ 0, & r > N. \end{cases} \tag{C.1}$$

These subspaces are encapsulated in the Grasmannian manifold $\mathrm{Gr}(r, N, \mathbb{F}_q)$ with $\dim(\mathrm{Gr}) = r(N-r)$.

$N\backslash r$	0	1	2	3	4	5	6	7	8	Σ
1	1	1	0	0	0	0	0	0	0	2
2	1	3	1	0	0	0	0	0	0	5
3	1	7	7	1	0	0	0	0	0	16
4	1	15	35	15	1	0	0	0	0	67
5	1	31	155	155	31	1	0	0	0	374
6	1	63	651	1395	651	63	1	0	0	2825
7	1	127	2667	11811	11811	2667	127	1	0	29212
8	1	255	10795	97155	200787	97155	10795	255	1	417199

Table C.1: Total number of r-dimensional subspaces of $\mathbb{F}_2^N$.

The total number of subspaces is obtained with $\sum_{r=0}^{N} \begin{bmatrix} N \\ r \end{bmatrix}_q$. This count gives also the so-called simplexity of the N-cube in a minimal corner-cut triangulation of the cube, which is geometrically intuitive. For $q = 2$ and $N = 1, \ldots, 8$, these are listed in Table C.1. Note that r starts from zero.

C.2 Pileup combinatorics

Let $M : \Theta \to Y$ be a probabilistic or stochastic mixing matrix

$$\mathbf{y} = M(\mathbf{p}, \mu)\mathbf{p}, \quad \sum_i M_{ij} = 1 \ \ \forall j \tag{C.2}$$

which has each column with unit sum and size $2^N - 1$. Each matrix element

$$M_{ij} \equiv P(c = i | c = j) \tag{C.3}$$

gives the conditional probability of an event vector originating from the j-th final state class c propagating to the i-th class, due to pileup. The matrix is thus interpreted vertically, which makes it a left stochastic matrix. This matrix is illustrated in Figure C.1.

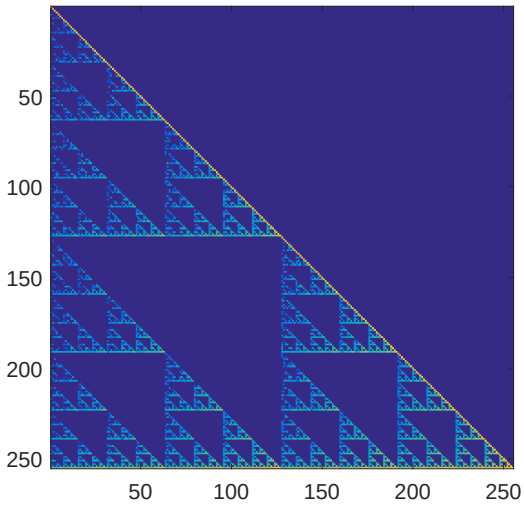

Figure C.1: Simulated pileup mixing matrix $M_{255\times255}$ for $N = 8$ and $\mu = 1.5$.

The matrix was obtained by a toy Monte Carlo pileup simulation using Poisson random numbers with a mean μ and by generating different final states according to uniform probabilities $p_c \sim 1/n$. The mixing matrix was obtained by counting the mixed final states event by event. Due to the permutation ambiguity, the originating class c was selected always to be the first of the random list, thus randomly. Because

the matrix M here is a function of the (unknown) $\mathbf{p}$, and also μ, it is not directly useful as an algorithmic inverse solution. Its function is to demonstrate algebraic properties. In general, the combinatorial enumerations can be approached by the 'twelve-fold way', which is a systematic approach developed by G.C. Rota. Here we list some combinatorial aspects of this problem to make it more transparent.

The number of non-zero elements in the pile-up mixing matrix per row are given by

$$a(i) = \sum_{r=1}^{i} \binom{i}{r} \bmod 2, \tag{C.4}$$

where mod denotes here the remainder after division and i is the index of the row. The sequence for the first 15 rows is

$$1, 1, 3, 1, 3, 3, 7, 1, 3, 3, 7, 3, 7, 7, 15, \quad i = 1, \ldots, 15. \tag{C.5}$$

Similarly, the number of 1 in the binary vector representation, or the so-called Hamming weight, is given by adding one and taking base-2 logarithm of Equation C.4

$$a(i) = \log_2 \left(\sum_{r=1}^{i} \binom{i}{r} \bmod 2 + 1 \right). \tag{C.6}$$

As an example

$$1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, \quad i = 1, \ldots, 15. \tag{C.7}$$

This sequence, namely the position of 1 in this sequence, give us directly the binary vectors which undergo only autocompound. That is, their signature cannot be imitated by a linear combination of other vectors. An example of this is given in Table C.3, for example when $N = 3$, these are $c = 1$,2 and 4. These are the unit basis vectors of $\mathbb{F}_2^N$. We also see the 'fractal' structure, that is, increasing the dimension of binary space keeps the multiplicity structure from lower dimensions which is evident from Table C.3. This is clear given the Hamming weight sequence.

The number of different terms for $c = 3$ (or $5, 6, 9, \ldots$) follows from a sequence

$$a(k) = \binom{k+1}{2} + k - 1 = \left\{ {k+1 \atop k} \right\} + k - 1, \tag{C.8}$$

which generates a multiplicity sequence $1, 4, 8, 13, 19, 26, \ldots$ and the curly brackets on right give the Stirling partition number (Stirling number of the second kind). This is the base of all higher dimensional binary vector space sequences. All the higher order

$\#_{(N,k)}$	1	2	3	4	5	6
2	3	6	10	15	21	28
3	7	28	84	210	462	924
4	15	120	680	3060	11628	38760
5	31	496	5456	46376	324632	1947792
6	63	2016	43680	720720	9657648	109453344
7	127	8128	349504	11358880	297602656	6547258432
8	255	32640	2796160	180352320	9342250176	404830840960
9	511	130816	22369536	2874485376	296071993728	25462191460608
10	1023	523776	178956800	45902419200	9428356903680	1615391816163840

Table C.2: Combinatorial total multiplicities for $N \in [2, 10]$ and $k \in [1, 6]$.

multiplicity matrices are self repeating results of this sequence or identity sequence $1, 1, \ldots$, which follows the algebraic binary expansion ordering. An example of this is given in Table C.3.

The total combinatorial or multinomial multiplicity (multiset multiplicity) is

$$\#_{(N,k)} = \binom{(2^N - 1) + k - 1}{(2^N - 1) - 1} \tag{C.9}$$

which is tabulated in Table C.2 for a finite number of N and k values. The factorial growth is obvious for high Poisson orders and binary dimensions.

$N=2$	k					
c	1	2	3	4	5	6
1	1	1	1	1	1	1
2	1	1	1	1	1	1
3	1	4	8	13	19	26
Σ	3	6	10	15	21	28

$N=3$	k					
c	1	2	3	4	5	6
1	1	1	1	1	1	1
2	1	1	1	1	1	1
3	1	4	8	13	19	26
4	1	1	1	1	1	1
5	1	4	8	13	19	26
6	1	4	8	13	19	26
7	1	13	57	168	402	843
Σ	7	28	84	210	462	924

$N=4$	k					
c	1	2	3	4	5	6
1	1	1	1	1	1	1
2	1	1	1	1	1	1
3	1	4	8	13	19	26
4	1	1	1	1	1	1
5	1	4	8	13	19	26
6	1	4	8	13	19	26
7	1	13	57	168	402	843
8	1	1	1	1	1	1
9	1	4	8	13	19	26
10	1	4	8	13	19	26
11	1	13	57	168	402	843
12	1	4	8	13	19	26
13	1	13	57	168	402	843
14	1	13	57	168	402	843
15	1	40	400	2306	9902	35228
Σ	15	120	680	3060	11628	38760

Table C.3: Combinatorial multiplicities for each individual binary vector (unit-hypercube vertex) with $N \in [2, 4]$ and $k \in [1, 6]$.

C.3 Exclusive efficiency

A term *exclusive efficiency* is often used in the context of pileup and large pseudo-rapidity gap veto based measurements of central diffraction. The probability for an event to be exclusive in a bunch cross is simply obtained from the zero-truncated Poisson distribution

$$P_P^{>0}(k;\mu) = P(X = k|X > 0) = \frac{P_P(k = 1;\mu)}{1 - P_P(k = 0;\mu)} = \frac{\mu^k}{(e^\mu - 1)k!}. \tag{C.10}$$

Then setting $k = 1$, which is the probability of one visible interaction, gives us the value of interest $P_P^{>0}(k = 1;\mu) = \mu/(e^\mu - 1)$. This runs asymptotically to zero when $\mu \to \infty$. The exclusive efficiency should not be mixed with the fundamental probability of rapidity gap survival, such as random re-scattering or production of secondaries filling the rapidity gap in the elementary pp-interaction. Neither it should be mixed with signal selection efficiency or direct veto *in*efficiency, these are either simulation based or control sample based corrections. The signal selection efficiency is driven by tracking and trigger efficiency whereas veto inefficiency is driven by efficiency of (forward) detectors to see all soft particles. A signal efficiency loss is simply of type multiplicative efficiency correction, the veto inefficiency, on the other hand, can be either multiplicative or additive (subtractive background) correction.

One 'obvious' idea would be to extrapolate the visible distributions of veto detectors down to a zero multiplicity or zero energy deposit by assuming a certain shape for the distributions (such as negative binomial distribution or similar), to estimate the veto inefficiency from data. However, this approach relies on several assumptions.

C.4 Poisson pileup problem

Let us assume for now that the Poissonian fluctuations are fluctuations in the number of simultaneous pp-interactions. That is, driven purely by instantaneous luminosity of the accelerator. The parameter μ describes the mean of visible simultaneous pp-interactions. We highlight the word visible because $k = 0$ includes not just 'empty' or 'non-interacting' bunch crossings but also experimentally non-visible inelastic interactions, the so-called low mass diffraction processes. These are the non-perturbative Regge domain QCD processes. Also, the elastic interactions belong to this category with cross section ~ 25 % of the total pp-cross section, which is $\sigma_{tot}^{pp} \sim \mathcal{O}(100)$ mb at the LHC. The Poisson distribution comes from the law of *rare events* and is an approximation of the binomial distribution $\mathrm{Bin}(n, p)$, when the number of trials $n \to \infty$ and $np = \mu$, then $\mathrm{Bin}(n, p) \to \mathrm{Poi}(\mu)$. For proton bunches circulating the LHC, n approximately $\sim 10^{11}$ – the number of protons per bunch, and p is order of $\sim 1/n$.

This gives us a finite probability of the triggered bunch cross event to originate from different Poisson terms as

$$P_P(k \geq 1; \mu) = 1 - P_P(0; \mu) = P_P(1; \mu) + P_P(2; \mu) + P_P(3; \mu) + \ldots \tag{C.11}$$

That is the sum of probabilities to have one visible pp-interaction, two pp-interactions (first order pileup) and so on, with the 'zero-suppressed' or truncated mean value: $\mathbb{E}[K|K > 0] = \mu/(1 - e^{-\mu}) \geq 1$ and the variance $\mathrm{Var}[K|K > 0] = (\mu + \mu^2)/(1 - e^{-\mu}) - \mu^2/(1 - e^{-\mu})^2$, which both converge to μ at high values of μ. Experimentally, the Poisson distribution μ-value can be obtained from the mean trigger normalized rate $R \in [0, 1)$ by

$$\boxed{R \equiv 1 - P_P(0; \mu) = 1 - e^{-\mu} \Leftrightarrow \mu = -\ln(1 - R)}. \tag{C.12}$$

Clearly when $R \ll 1$, then $\mu \simeq R$. The mean normalized trigger rate R itself is obtained from the measured trigger frequency f_R (Hz) as a function of time t. This we get by using the accelerator orbital frequency $f_O = 11.2455$ kHz at the LHC and the number of colliding bunch pairs N_B circulating

$$\langle R \rangle_{t,N_B} = \frac{\frac{1}{\Delta t} \int_{\Delta t} f_R(t)\, dt}{N_B f_O} \quad \text{or} \quad \langle R \rangle_{N_B} = \frac{N_E}{N_{BC}}. \tag{C.13}$$

The interval Δt corresponds to the given data run span over time. The alternative version on right is just a reformulation using the number of triggered events N_E and the number of bunch crosses N_{BC}. For the formalism it is important to emphasize that the trigger corresponds to Boolean OR ($\vee$) operation between different

(sub)detectors, which is the minimum bias trigger. The reason is that this operator is the most inclusive Boolean operator.

In general, care is required in determining the μ-value experimentally because it is effectively a factor of both on-time and off-time pileup due to the detector time integration windows which can span over several bunch crosses. It must be emphasized that the formalism here requires only an *effective* μ-value of the Poisson distribution. For the rest of the work, we assume that the effective modeling and measurement of the μ-value is under control. The μ value determination at large μ is problematic directly from the $\mu = -\ln(1-R)$ due to $R \simeq 1$. However, under high luminosity, the μ value is measured with less saturated signals, by using for example track or primary vertex counting.

We assume that the pileup is linear concerning the detector responses. Thus, we can take a linear incoherent superposition of the final states propagating from different simultaneous pp-interactions. Also to point out explicitly: when we 'sum' binary vectors, we use the component-wise OR, for example: $[1,0] \vee [1,1] = [1,1] \in \mathbb{F}_2^2$.

C.5 Luminosity and total inelastic cross section

The μ-value can be understood also naturally in the context of instantaneous luminosity L ($\mathrm{cm}^{-2}\cdot\mathrm{s}^{-1}$) measurement by using a simplified definition as in [155]

$$\mu_{abs} \equiv \frac{\sigma_{inel} L}{N_B f_O}, \tag{C.14}$$

where σ_{inel} is the total inelastic cross section ($\sim$ 80 mb in pp at $\sqrt{s}$ = 13 TeV), dictated by non-perturbative strong interactions, N_B is the number of colliding bunch pairs and f_O is the LHC orbital frequency (Hz). The μ_{abs} is by definition the absolute mean number of inelastic interactions per colliding bunch cross.

However, because the complete σ_{inel} is essentially unknown due to limited low-mass diffraction acceptance at the LHC, also the μ_{abs} is unknown. The absolute mean number of inelastic interactions per bunch crossing μ_{abs}, is reduced to the measured μ by the total integrated detector multiplicative acceptance $\times$ efficiency factor $0 \leq \epsilon \leq 1$ defining[1] the visible inelastic cross section as

$$\sigma_{vis} \equiv \epsilon \sigma_{inel}. \tag{C.15}$$

Then using linearity of Eq. C.14 with respect to the cross section, we can write

$$\frac{\mu}{\mu_{abs}} = \frac{\sigma_{vis}}{\sigma_{inel}} \tag{C.16}$$

and reformulating

$$L = \mu_{abs} \frac{N_B f_O}{\sigma_{inel}} = \mu \frac{N_B f_O}{\sigma_{vis}}. \tag{C.17}$$

Now L or equivalently σ_{vis}, can be measured by doing a van der Meer (vdM) scan, which is an absolute luminosity measurement calibration technique. However, the 'total inelastic measurement' is based on an extrapolation inverting Eq. C.15. The efficiency $\times$ acceptance factor ϵ here has an uncertainty at least order of $\delta\epsilon/\epsilon \sim 0.1$. Also, a crucial thing is to factorize the acceptance and efficiency, but experimentally one cannot always distinguish between these two and this creates the problem of defining the fiducial acceptance domain. This is because not all particles and their momentum of multibody decays of systems are measured, thus, the efficiency and acceptance have intrinsic detector simulation and minimum bias soft QCD Monte Carlo model dependence.

[1] Technically, we may have also 'leakage' of events from outside the geometric acceptance, which do not exactly obey this definition, but nevertheless the effective ϵ is never larger than 1.

Appendix C. Combinatorics and pileup

It is a significant challenge for experiments to carefully define the final state observables in the forward domain. Fiducial definitions may be in terms of single particle acceptance, described by (η, p_t) or $|\vec{p}|$, or in terms of the (diffractive) system invariant mass. Intuitively, the fiducial definition in terms of single particle kinematics should be less model-dependent than definitions relying on the invariant mass, at least if momentum or energy measurements are available. A special case to the discussion here is an indirect inference based on the elastic scattering, *extrapolation* of $d\sigma_{el}/dt$ down to $t \to 0$ and using the optical theorem (unitarity) relating the total cross section and the imaginary part of the elastic forward amplitude.

C.6 F^*-projection technique

We shall here shortly outline a data-MC hybrid interpolation and projection technique, what we may call the F^*-projection. The idea is simple: observables which cannot be directly reconstructed with the detector, may be multidimensionally interpolated using the Monte Carlo event generator based simulations, given the measured partial cross sections. This can be done at the generator level once the measured partial cross sections have been unfolded. This is closely related to non-orthogonal and overcomplete basis projection techniques known in the Wavelet and Compressed Sensing literature, for those see [235]. In Monte Carlo, for each $j = 1, \ldots, n = 2^N - 1$ combinatorially selected sub-sample, we construct the probability distribution $f^{\mathrm{MC}}(\mathbf{O})_{|j}$ of the observable $\mathcal{O}$. We may use normalized histograms to represent the distributions, for example. This gives us a set

$$\left\{ f^{\mathrm{MC}}(\mathcal{O})_{|j} \right\}_{j=1}^{n} . \tag{C.18}$$

Then we measure the set of combinatorial partial cross sections in data

$$\left\{ \sigma_j^{\mathrm{DATA}} \right\}_{j=1}^{n} . \tag{C.19}$$

The F^*-projected estimate of the differential cross section is obtained with

$$\frac{d\sigma^{F^*}}{d\mathcal{O}} = \sum_j \sigma_j^{\mathrm{DATA}} f^{\mathrm{MC}}(\mathcal{O})_{|j}, \tag{C.20}$$

which shows clearly that the result is a sum over Monte Carlo driven 'dictionary functions' weighted with data driven coefficients. It is possible to show that for certain observables, when $N \to \infty$, the dependence on Monte Carlo is minimized. In practice, one wants to make the projection using several different Monte Carlo samples to obtain estimate of the model dependence. Natural distributions to reconstruct using this technique are rapidity gap size distributions $d\sigma/d\Delta y$ with varying boundary conditions, for example.

C.7 Diffraction fit algorithms

Let us have $\mathcal{C}$ different scattering processes for which we have a Monte Carlo event generator based predictions of the multidimensional combinatorial partial cross sections, normalized to probability densities. We write these down as rows in a model likelihood matrix $\mathcal{L}$ with size $\mathcal{C} \times 2^N - 1$. Now given the measured partial cross sections, we want to obtain the Maximum Marginal Likelihood solution: the best fit of the measurement as a weighted incoherent sum of the Monte Carlo model based 'multidimensional template distributions'. The solution to this is obtained via Expectation Maximization iteration given in Algorithm 4. If in addition, we want to re-weight the event generator distributions to simultaneously fit e.g. the effective Pomeron intercept, we proceed with Algorithm 5. Statistical fit uncertainties can be obtained via bootstrap re-sampling. These algorithms can be used either at unfolded fiducial cross section or at visible (detector) level.

Algorithm 4 Maximum Marginal Likelihood estimator via EM-iteration.

INPUT: (Unfolded) measurement vector $\boldsymbol{\sigma}$ $(2^N - 1 \times 1)$, model likelihood matrix $\mathcal{L}$ $(\mathcal{C} \times 2^N - 1)$ with $\forall i : \sum_j [\mathcal{L}]_{i,j} = 1$, initial estimate Θ $(\mathcal{C} \times 1)$, otherwise $\Theta = \mathbf{1}_{\mathcal{C}}/|\mathcal{C}|$.

repeat
A. Inverse step using Bayes theorem:
Diagonal priors matrix $\times$ Likelihood matrix:
$\mathcal{P} \leftarrow \mathrm{diag}(\Theta_k)\mathcal{L}$
Normalize each column to obtain the probabilistic mixed density operator:
for all $j = 1, \ldots, 2^N - 1$ **do**
$[\mathcal{P}]_{:,j} \leftarrow [\mathcal{P}]_{:,j} / \sum_{i=1}^{\mathcal{C}} [\mathcal{P}]_{i,j}$
end for
B. Forward step:
Operate on data with the mixed density operator:
$\Theta_{k+1} \leftarrow \mathcal{P}\boldsymbol{\sigma}$
until Convergence $\|\Theta_{k+1} - \Theta_k\|_2 < \delta$

OUTPUT: The process cross section estimates $\hat{\Theta} = (\sigma^1, \sigma^2, ..., \sigma^{\mathcal{C}})^T$.

Algorithm 5 Event-by-event MC re-weighting via M parameter brute force grid scan.

INPUT:

for all M parameters in $\Phi = [\phi_a^1, \phi_b^1] \times [\phi_a^2, \phi_b^2] \times \cdots \times [\phi_a^M, \phi_b^M]$ **do**

for all MC events **do**

Construct the MC observables functionally dependent on the parameters

Re-weight the event according to $(\phi^1, \phi^2, \ldots, \phi^M)$

end for

A. Construct a re-weighted MC process Likelihood matrix with weighted event selection

B. Estimate the process cross sections (mixing weights) using Algorithm A4

$\hat{\Theta} \leftarrow$A4$(\tilde{\boldsymbol{\sigma}}, \mathcal{L}_{RW})$

C. Construct the new Monte Carlo estimate of $2^N - 1$ partial cross sections

$\tilde{\boldsymbol{\sigma}}^{MC} \leftarrow \mathcal{L}_{RW}^T \hat{\Theta}$

D. Calculate Kullback-Leibler divergence D(data|model) with

$D \leftarrow D(\tilde{\boldsymbol{\sigma}} | \boldsymbol{\sigma}^{MC})$

end for

OUTPUT: Kullback-Leibler divergences for the parameter (hyper)grid $\Phi \subseteq \mathbb{R}^M$. Use these to infer optimal values and parameter sensitivity/uncertainty.

D Graniitti Code (2×2)

D.1 Makefile

Makefile 1/6

```
# GRANIITTI Makefile
#
# (c) 2017-2020 Mikael Mieskolainen.
# Licensed under the MIT License <http://opensource.org/licenses/MIT>.
# ---------------------------------------------------------------------
#
# COMPILING:
#       make -j4
#
# CLEANING:
#       make clean (objects)
#       make superclean (objects + binaries)
#
# To compile with clang:                make CXX=clang
#     -|-           unit tests:         make TEST=TRUE
#
# To compile with ROOT using -std=c++14:   make ROOT14=TRUE
#
# To compile with profiling:            make VALGRIND=TRUE
#
# ---------------------------------------------------------------------
#
# EXTERNAL LIBRARIES SETUP:
#
#   [HEPMC3] and [LHAPDF]: See ./install folder
#   [ROOT]                 Example: export ROOTSYS=/home/user/sw/ROOT6
#
# GENERAL:
#
#   If you just installed e.g. HEPMC3 libraries
#   $> sudo ldconfig
#
# ---------------------------------------------------------------------
#
# If you see symbol errors related to HepMC such that:
# "undefined symbol: _ZN5HepMC10FilterBase19init_has_end_vertexEv"
# check that you do not have a collision of HEPMC3 with an existing
# HepMC2 shared library installation!
#
# Check below:
#
# For tracking the libraries of the program:
#  $> ldd ./executable
#
# Check if the shared object (.so) file contains the missing symbol:
#  $> nm -D libHepMC.so
#
# ---------------------------------------------------------------------
# Creating shared libraries, use -fPIC flag, then:
# g++ -shared MH1.o -o libMH1.so
#
#
# USE [TABS] for intendation while modifying this file!
# ---------------------------------------------------------------------


# ======================================================================
# Detect compiler version if using by default g++

ifeq ($(CXX),g++)

# KERNEL_GCC_VERSION
# := $(shell cat /proc/version \
# | sed -n 's/.*gcc version \([[:digit:]]\.[[:digit:]]\.[[:digit:]]\).*/\1/p')
# $(info Info KERNEL_GCC_VERSION = $(KERNEL_GCC_VERSION))

CXX_VERSION = $(shell g++ -dumpversion)
$(info Found CXX_VERSION = $(CXX_VERSION))
CXX_REQUIRED = 7.0

# Use bc command for double comparison, a common utility on Unix platforms
FOO=$(shell echo "$(CXX_VERSION) < $(CXX_REQUIRED)" | bc)

ifeq ($(FOO),1)
$(error Your CXX_VERSION is too old, need at least CXX_VERSION = $(CXX_REQUIRED), update your GCC chain)
endif

endif

# ======================================================================
# PATH setup

# HEPMC3 installation
```

Makefile 2/6

```
#HEPMC3SYS      = $(shell HepMC-config --prefix)

# LHAPDF installation
#LHAPDFSYS      = $(shell lhapdf-config --prefix)

# ROOT installation
#ROOTSYS        = $(shell root-config --prefix)


ifeq ($(HEPMC3SYS),)
$(error Please set HEPMC3SYS environment variable and paths [perhaps 'source ./install/setenv.sh'])
else
$(info Found HEPMC3SYS = $(HEPMC3SYS))
endif

ifeq ($(LHAPDFSYS),)
$(error Please set LHAPDFSYS environment variable and paths [perhaps 'source ./install/setenv.sh'])
else
$(info Found LHAPDFSYS = $(LHAPDFSYS))
endif

# If ROOT seems to be installed, tag ROOT dependent compilation on
ifeq ($(ROOTSYS),)
$(info Did not find ROOTSYS - compilation without ROOT <root.cern.ch> dependent tools)
ROOT=FALSE
else
$(info Found ROOTSYS = $(ROOTSYS))
ROOT=TRUE
endif

# Info messages
$(info )
$(info ********************************************************************************)
$(info **                                                                            **)
$(info **   If compilation with ROOT libraries failes, try with: make -j4 ROOT14=TRUE    **)
$(info **                                                                            **)
$(info ********************************************************************************)
$(info )
endif


# ======================================================================
# External libraries to be linked

# HEPMC3  (lib64 needed on some systems)
HEPMC3lib      = -L$(HEPMC3SYS)/lib -L$(HEPMC3SYS)/lib64 -lHepMC3 -lHepMC3search

# LHAPDF6 (lib64 needed on some systems)
LHAPDF61lib    = -L$(LHAPDFSYS)/lib -L$(LHAPDFSYS)/lib64 -lLHAPDF

# PyTorch
# Note -Wl,-rpath-link= handles the recursive dependency (for linker)
#PYTORCH1lib     = -L./libs/libtorch/lib -lc10 -ltorch -lgomp -lcaffe2 \
#                                        -Wl,-rpath-link=./libs/libtorch/lib

# ROOT
ifeq ($(ROOT),TRUE)
ROOT1lib       = -L$(ROOTSYS)/lib -L$(ROOTSYS)/lib/root -lCore -lRIO -lNet \
                                        -lHist -lGraf -lGraf3d -lGpad -lTree -lRint \
                                        -lPostscript -lMatrix -lPhysics -lMathCore \
                                        -lThread -lGui -lRooFit -lMinuit
endif

# C++ standard
STANDARD1lib   = -lstdc++ -lm -pthread -lrt
#  -ldl -rdynamic

# GSL, on ubuntu run: sudo apt-get install libgsl-dev
#GSL1lib         = -lgsl -lgslcblas


# External libraries (THESE TWO FIRST for priority!!)
LDLIBS  = $(HEPMC3lib)
LDLIBS += $(LHAPDF61lib)

# The rest
LDLIBS += $(STANDARD1lib)
#LDLIBS += $(PYTORCH1lib)


# ======================================================================
# Header files

# External libraries (THESE TWO FIRST for priority!!)
```

Makefile 3/6

```
INCLUDES += -I$(HEPMC3SYS)/include
INCLUDES += -I$(LHAPDFSYS)/include

# C++
INCLUDES += -I/usr/include
INCLUDES += -I/usr/local/include

# Own
INCLUDES += -I.
INCLUDES += -Iinclude

# Internal libraries
INCLUDES += -Ilibs

# FTensor
INCLUDES += -Ilibs/FTensor

# Eigen
INCLUDES += -Ilibs/Eigen/unsupported/

# PyTorch
#INCLUDES += -Ilibs/libtorch/include/
#INCLUDES += -Ilibs/libtorch/include/torch/csrc/api/include


# ======================================================================
# Compiler and its options

CXX         = g++

CXXVER      = -std=c++17
CXXVER_OLD = -std=c++17

# Use this for alternative ROOT installations
ifeq ($(ROOT14),TRUE)
CXXVER_OLD = -std=c++14
endif

OPTIM       = -O3 -DNDEBUG -ftree-vectorize -fno-signed-zeros
CXXFLAGS    = -Wall -fPIC -pipe $(OPTIM)

# Profiling & debug
ifeq ($(VALGRIND),TRUE)
OPTIM   += -pg
endif


# Needed by PyTorch if using pre-compiled (ABI = Application Binary Interface)
# gcc < 5.1 is 0, later versions use 1 by default
# CXXFLAGS += -D_GLIBCXX_USE_CXX11_ABI=0

# Automatic dependencies on
CXXFLAGS += -MMD -MP


# -Wall,              compiler warnings full on
# -free-vectorize,    Autovectorization on
# -fno-signed-zeros   Optimization (floating point) which ignore the signedness of zero
# -fPIC,              Position independent code (PIC) for shared libraries
# -O2, -O3            Optimization level
# -DNDEBUG            Release flag, no debug
# -pipe,              Faster compilation using pipes
# -O0 -pg,            Profiling and debugging (HUGE PERFORMANCE HIT)
# -rpath-link         Needed for .so which links to another .so
#
# Check your CPU instruction set with: cat /proc/cpuinfo


# DANGEROUS:

# -march=native,     CPU spesific instruction set usage
#  (gives unknown flops problem, factor 1/4 wrong results on i5-4570 with g++7.4, do not use!)
# -ffast-math,       Heavy floating point optimization
#  (fast but breaks IEEE flops standards, do not use!)

# ======================================================================
# Sources, objects and dependency files

OBJ_DIR     = obj


# ---------------------------------------------------------------------
```

Makefile 4/6

```
# LIBRARY

## ======================================================================
SRC_DIR_0   = src
SRC_0       = $(wildcard $(SRC_DIR_0)/*.cc)
OBJ_0       = $(SRC_0:$(SRC_DIR_0)/%.cc=$(OBJ_DIR)/%.o)
DEP_0       = $(OBJ_0:$(OBJ_DIR)/%.o=.d)
## ======================================================================

## ======================================================================
SRC_DIR_1   = src/Amplitude
SRC_1       = $(wildcard $(SRC_DIR_1)/*.cc)
OBJ_1       = $(SRC_1:$(SRC_DIR_1)/%.cc=$(OBJ_DIR)/%.o)
DEP_1       = $(OBJ_1:$(OBJ_DIR)/%.o=.d)
## ======================================================================

# Create collection of all library objects
OBJ = $(OBJ_0) $(OBJ_1)

# ======================================================================
ifeq ($(ROOT),TRUE)
SRC_DIR_2   = src/Analysis
SRC_2       = $(wildcard $(SRC_DIR_2)/*.cc)
OBJ_2       = $(SRC_2:$(SRC_DIR_2)/%.cc=$(OBJ_DIR)/%.o)
DEP_2       = $(OBJ_2:$(OBJ_DIR)/%.o=.d)
endif
# ======================================================================

# ======================================================================
ifeq ($(TEST),TRUE)
SRC_DIR_3   = tests/catchlibrary
SRC_3       = $(wildcard $(SRC_DIR_3)/*.cc)
OBJ_3       = $(SRC_3:$(SRC_DIR_3)/%.cc=$(OBJ_DIR)/%.o)
DEP_3       = $(OBJ_3:$(OBJ_DIR)/%.o=.d)
endif
# ======================================================================


# ----------------------------------------------------------------------
# PROGRAM compiled against the library

# ======================================================================
SRC_DIR_PROGRAM      = src/Program
SRC_PROGRAM          = $(wildcard $(SRC_DIR_PROGRAM)/*.cc)
OBJ_PROGRAM          = $(SRC_PROGRAM:$(SRC_DIR_PROGRAM)/%.cc=$(OBJ_DIR)/$(BIN_DIR)/%.o)
DEP_PROGRAM          = $(OBJ_PROGRAM:$(OBJ_DIR)/$(BIN_DIR)/%.o=.d)
# ======================================================================

# ======================================================================
ifeq ($(TEST),TRUE)
SRC_DIR_PROGRAM_ROOT = src/Program/Analysis
SRC_PROGRAM_ROOT     = $(wildcard $(SRC_DIR_PROGRAM_ROOT)/*.cc)
OBJ_PROGRAM_ROOT     = $(SRC_PROGRAM_ROOT:$(SRC_DIR_PROGRAM_ROOT)/%.cc=$(OBJ_DIR)/$(BIN_DIR)/%.o)
DEP_PROGRAM_ROOT     = $(OBJ_PROGRAM_ROOT:$(OBJ_DIR)/$(BIN_DIR)/%.o=.d)
endif
# ======================================================================

# ======================================================================
ifeq ($(TEST),TRUE)
SRC_DIR_PROGRAM_TEST = tests
SRC_PROGRAM_TEST     = $(wildcard $(SRC_DIR_PROGRAM_TEST)/*.cc)
OBJ_PROGRAM_TEST     = $(SRC_PROGRAM_TEST:$(SRC_DIR_PROGRAM_TEST)/%.cc=$(OBJ_DIR)/$(BIN_DIR)/%.o)
DEP_PROGRAM_TEST     = $(OBJ_PROGRAM_TEST:$(OBJ_DIR)/$(BIN_DIR)/%.o=.d)
endif
# ======================================================================


# ----------------------------------------------------------------------
# PROGRAM

# Directory
BIN_DIR = bin

.SUFFIXES:      .o .cc

# Normal
EXE_NAMES      = gr xscan minbias hepmc3tolhe data2hepmc3 pathmark pdebench sommerfeld ot
PROGRAM        = $(EXE_NAMES:%=$(BIN_DIR)/%)

ifeq ($(ROOT),TRUE)
EXE_ROOT_NAMES = analyze fitsoft fitcentral fitharmonic
```

```
PROGRAM_ROOT   = $(EXE_ROOT_NAMES:%=$(BIN_DIR)/%)
endif

PROGRAM_TEST   =
ifeq ($(TEST),TRUE)
EXE_TEST_NAMES = testbench0 testbench1 testbench2 testbench3
PROGRAM_TEST   = $(EXE_TEST_NAMES:%=$(BIN_DIR)/%)
endif

# Multicore
#export MAKEFLAGS="-j $(grep -c ^processor /proc/cpuinfo)"

# ------------------------------------------------------------------------
# RULES for linking

all: $(PROGRAM) $(PROGRAM_ROOT) $(PROGRAM_TEST)
        @echo " "
        @echo "PROGRAM:" $(PROGRAM) $(PROGRAM_ROOT) $(PROGRAM_TEST)
        @echo " "
        @echo "Compilation of '$@' done."

$(PROGRAM) : $(OBJ) $(OBJ_PROGRAM)
        $(CXX) $(OBJ_DIR)/$@.o $(OBJ) -o $@ $(CXXFLAGS) $(LDLIBS)

$(PROGRAM_ROOT) : $(OBJ) $(OBJ_2) $(OBJ_PROGRAM_ROOT)
        $(CXX) $(OBJ_DIR)/$@.o $(OBJ) $(OBJ_2) -o $@ $(CXXFLAGS) $(LDLIBS) $(ROOTlib)

# Unit tests (note, we use catchmain.o from $(OBJ_3) for linking with catch2)
$(PROGRAM_TEST) : $(OBJ) $(OBJ_3) $(OBJ_PROGRAM_TEST)
        $(CXX) $(OBJ_DIR)/$@.o $(OBJ) $(OBJ_3) -o $@ $(CXXFLAGS) $(LDLIBS)


# ------------------------------------------------------------------------
# RULES to generate library dependencies and compile

# LIBRARY objects
#
# Note that for different ROOT installations, we need both:
# -I$(ROOTSYS)/include
# -I$(ROOTSYS)/include/root

# ========================================================================
$(OBJ_DIR)/%.o: $(SRC_DIR_0)/%.cc
        @echo " "
        @echo "Generating dependencies and compiling $<..."
        $(CXX) $(CXXVER) -c $< -o $@ $(CXXFLAGS) $(INCLUDES)
# ========================================================================

# ========================================================================
$(OBJ_DIR)/%.o: $(SRC_DIR_1)/%.cc
        @echo " "
        @echo "Generating dependencies and compiling $<..."
        $(CXX) $(CXXVER) -c $< -o $@ $(CXXFLAGS) $(INCLUDES)
# ========================================================================

# ========================================================================
$(OBJ_DIR)/%.o: $(SRC_DIR_2)/%.cc
        @echo " "
        @echo "Generating dependencies and compiling $<..."
        $(CXX) $(CXXVER_OLD) -c $< -o $@ $(CXXFLAGS) $(INCLUDES) \
                -I$(ROOTSYS)/include -I$(ROOTSYS)/include/root
# ========================================================================

# ========================================================================
$(OBJ_DIR)/%.o: $(SRC_DIR_3)/%.cc
        @echo " "
        @echo "Generating dependencies and compiling $<..."
        $(CXX) $(CXXVER) -c $< -o $@ $(CXXFLAGS) $(INCLUDES)
# ========================================================================

# PROGRAM objects

# ========================================================================
$(OBJ_DIR)/$(BIN_DIR)/%.o: $(SRC_DIR_PROGRAM)/%.cc
        @echo " "
        @echo "Generating dependencies and compiling $<..."
        $(CXX) $(CXXVER) -c $< -o $@ $(CXXFLAGS) $(INCLUDES)
# ========================================================================

# ========================================================================
$(OBJ_DIR)/$(BIN_DIR)/%.o: $(SRC_DIR_PROGRAM_ROOT)/%.cc
        @echo " "
```



```
        @echo "Generating dependencies and compiling $<..."
        $(CXX) $(CXXVER_OLD) -c $< -o $@ $(CXXFLAGS) $(INCLUDES) \
                -I$(ROOTSYS)/include -I$(ROOTSYS)/include/root
# ========================================================================

# ========================================================================
$(OBJ_DIR)/$(BIN_DIR)/%.o: $(SRC_DIR_PROGRAM_TEST)/%.cc
        @echo " "
        @echo "Generating dependencies and compiling $<..."
        $(CXX) $(CXXVER) -c $< -o $@ $(CXXFLAGS) $(INCLUDES)
# ========================================================================

# ------------------------------------------------------------------------
# Clean up of object files
# - ignores return code error
# @ is silent
.PHONY: clean
clean:
        @echo "Cleaning objects"
        -@rm $(OBJ_DIR)/*.o 2>/dev/null || true
        -@rm $(OBJ_DIR)/*.d 2>/dev/null || true
        -@rm $(OBJ_DIR)/$(BIN_DIR)/*.o 2>/dev/null || true
        -@rm $(OBJ_DIR)/$(BIN_DIR)/*.d 2>/dev/null || true

.PHONY: superclean
superclean: clean
        @echo "Cleaning binaries"
        -@rm $(BIN_DIR)/* 2>/dev/null || true

# ------------------------------------------------------------------------
# Dependencies listed here
-include $(DEP_0)
-include $(DEP_1)
ifeq ($(ROOT),TRUE)
-include $(DEP_2)
endif
ifeq ($(TEST),TRUE)
-include $(DEP_3)
endif

# DEBUG printing

#print_vars:
#       echo FOO = $(OBJ)
```

D.2 MadGraph amplitude to Graniitti converter

./develop/MG2GRA/MG2GRA.py 1/3

```
# Python processor for MadGraph C++ amplitude output to GRANIITTI conversion
#
# The input folder should contain files:
# CPPProcess.cc
# CPPProcess.h
# param_card.dat
#
# Example with folder 'yy_ll':     yes yy_ll | python MG2GRA.py
#
# (c) 2017-2020 Mikael Mieskolainen
# Licensed under the MIT License <http://opensource.org/licenses/MIT>.

# Set here the new process name
PROCESS = input("Give the name of the process folder: ")
#PROCESS = 'gg_gg'


# ------------------------------------------------------------------------
CLASSNAME = 'AMP_MG5_' + PROCESS;

message_orig = '// Visit launchpad.net/madgraph5 and amcatnlo.web.cern.ch'
message_new  = ('// Visit launchpad.net/madgraph5 and amcatnlo.web.cern.ch'
                                '\n// @@@@ MadGraph to GRANIITTI conversion done @@@@')

# ------------------------------------------------------------------------
# Process the header file

fin  = open('./' + PROCESS + '/' + 'CPPProcess.h', 'rt')
fout = open('./output/' + CLASSNAME + '.h', 'wt')

for line in fin:

        line = line.replace('virtual','')
        line = line.replace(message_orig, message_new)

        line = line.replace('#include "Parameters_sm.h',
                ('#include "Graniitti/Amplitude/Parameters_sm.h"'
                 '\n#include "Graniitti/MForm.h"'
                 '\n#include "Graniitti/MAux.h"'
                 '\n#include "Graniitti/MKinematics.h"'
                 '\n#include "Graniitti/MMath.h"'))

        line = line.replace('CPPProcess', CLASSNAME)
        line = line.replace('Parameters_sm * pars;', 'Parameters_sm pars; // GRANIITTI')
        line = line.replace('void sigmaKin()', 'double CalcAmp2(gra::LORENTZSCALAR &lts, double alphas)')

        line = line.replace(CLASSNAME + '() {}',
                CLASSNAME + '() {initProc(gra::aux::GetBasePath(2) + "/MG5cards/param_card_'
                + PROCESS + '.dat"); }');

        fout.write(line)

fin.close()
fout.close()

print('Output to ' + CLASSNAME + '.h created')

# ------------------------------------------------------------------------
# Process the source file

fin  = open('./' + PROCESS + '/' + 'CPPProcess.cc', 'rt')
fout = open('./output/' + CLASSNAME + '.cc', 'wt')


for line in fin:

        line = line.replace(message_orig, message_new)

        line = line.replace('#include "HelAmps_sm.h', '#include "Graniitti/Amplitude/HelAmps_sm.h')
        line = line.replace('#include "CPPProcess.h', '#include "Graniitti/Amplitude/CPPProcess.h')

        line = line.replace('void CPPProcess::sigmaKin()',
                'double CPPProcess::CalcAmp2(gra::LORENTZSCALAR &lts, double alphas)')
        line = line.replace('CPPProcess', CLASSNAME)

        line = line.replace('pars = Parameters_sm::getInstance();','pars = Parameters_sm(); // GRANIITTI')
        line = line.replace('pars->', 'pars.')
        line = line.replace('static bool firsttime = true;','static bool firsttime = false; // GRANIITTI')
        line = line.replace('ntry = ntry + 1;','ntry = 1; // GRANIITTI')

        line = line.replace('setDependentParameters()', 'setDependentParameters(alphas)')
```

./develop/MG2GRA/MG2GRA.py 2/3

```

        s = line.find('printIndependentParameters()')
        if s != -1:
                line = '// pars.printIndependentParameters(); // GRANIITTI \n'

        s = line.find('printIndependentCouplings()')
        if s != -1:
                line = '// pars.printIndependentCouplings(); // GRANIITTI \n'

        s = line.find('setDependentCouplings()')
        if s != -1:
                extra = 'if (gra::PARAM_STRUCTURE::QED_alpha == "ZERO") \n'
                fout.write(extra)
                extra = 'pars.setAlphaQEDZero(); // GRANIITTI: Set at scale alpha_QED(Q2=0) \n\n'
                fout.write(extra)

        s = line.find('if (sum_hel == 0')
        if s != -1:
                extra = 'goto SKIPLABEL; // GRANIITTI: Skip this block \n'
                fout.write(extra)

        fout.write(line)

        # ------------------------------------------------------------------
        # Find the first lines of the main calculation function
        s = line.find('// Local variables and constants')
        blockline = """
        // @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ GRANIITTI @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
        // *** MADGRAPH CONVENTION IS [E,px,py,pz] ! ***

        // *** Set masses for HELAS ***
        std::vector<double> masses = {0, 0}; // Massless two initial states
        std::vector<gra::M4Vec> pf;

        for (std::size_t i = 0; i < lts.decaytree.size(); ++i) {
                masses.push_back(lts.decaytree[i].p4.M());
                pf.push_back(lts.decaytree[i].p4);
        }
        mME = masses;

        gra::M4Vec p1_ = lts.q1;
        gra::M4Vec p2_ = lts.q2;


        // Do kinematic transform
        gra::kinematics::OffShell2LightCone(p1_, p2_, pf);

        // Set initial state 4-momentum
        p.clear();
        double p1[] = {p1_.E(), p1_.Px(), p1_.Py(), p1_.Pz()};
        p.push_back(&p1[0]);

        double p2[] = {p2_.E(), p2_.Px(), p2_.Py(), p2_.Pz()};
        p.push_back(&p2[0]);

        // Set final state 4-momentum
        double pthis[pf.size()][4];
        for (std::size_t i = 0; i < pf.size(); ++i) {
                pthis[i][0] = pf[i].E();
                pthis[i][1] = pf[i].Px();
                pthis[i][2] = pf[i].Py();
                pthis[i][3] = pf[i].Pz();
                p.push_back(&pthis[i][0]);
        }
        // @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ GRANIITTI @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
        """
        if s != -1:
                fout.write(blockline)
                continue;

        # ------------------------------------------------------------------
        # Number of combinations
        s = line.find('const int ncomb =')

        if s != -1:
                extra = 'lts.hamp = std::vector<std::complex<double>>(ncomb); // GRANIITTI \n\n'
                fout.write(extra)
                continue
        # ------------------------------------------------------------------

        # Find the last line of the main calculation function
```

./develop/MG2GRA/MG2GRA.py 3/3

```
        s = line.find('matrix_element[i] /= denominators[i];')
        blockline = """
        SKIPLABEL:

        // @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ GRANIITTI @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
        // Define permutation
        for (int i = 0; i < nexternal; ++i) { perm[i] = i; }

        // Loop over helicity combinations
        for (int ihel = 0; ihel < ncomb; ++ihel) {
                calculate_wavefunctions(perm, helicities[ihel]);

                // Sum of subamplitudes (s,t,u,...)
                for (int k = 0; k < namplitudes; ++k) { lts.hamp[ihel] += amp[k]; }
        }

        // Total amplitude squared over all helicity combinations individually
        double amp2 = 0.0;
        for (int ihel = 0; ihel < ncomb; ++ihel) {
        amp2 += gra::math::abs2(lts.hamp[ihel]);
        }
        amp2 /= denominators[0];  // spin average matrix element squared

        return amp2;  // amplitude squared
        // @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ GRANIITTI @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
        """

        if s != -1:
                fout.write(blockline)

fin.close()
fout.close()

print('Output to ' + CLASSNAME + '.cc created')


# ------------------------------------------------------------------------
# Copy parameter card

import os
os.system('cp ./' + PROCESS + '/param_card.dat ./output/' + 'param_card_' + PROCESS +'.dat')
print('Output to param_card_' + PROCESS + '.dat created')


os.system('cp ./output/*.h ../../include/Graniitti/Amplitude/')
os.system('cp ./output/*.cc ../../src/Amplitude/')
os.system('cp ./output/*.dat ../../MG5cards/')
```

D.3 C++ header files

./include/Graniitti/MQuasiElastic.h 1/1

```
 1: // QuasiElastic (EL,SD,DD) and soft ND class
 2: //
 3: // (c) 2017-2020 Mikael Mieskolainen
 4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
 5:
 6: #ifndef MQUASIELASTIC_H
 7: #define MQUASIELASTIC_H
 8:
 9: // C++
10: #include <complex>
11: #include <random>
12: #include <vector>
13:
14: // HepMC33
15: #include "HepMC3/GenEvent.h"
16:
17: // Own
18: #include "Graniitti/M4Vec.h"
19: #include "Graniitti/MAux.h"
20: #include "Graniitti/M4Matrix.h"
21: #include "Graniitti/MProcess.h"
22: #include "Graniitti/MSpin.h"
23:
24: namespace gra {
25: // Matrix element dimension: " GeV^" << -(2*external_legs - 8)
26: class MQuasiElastic : public MProcess {
27:  public:
28:   MQuasiElastic();
29:   MQuasiElastic(std::string process, const std::vector<aux::OneCMD> &syntax);
30:   virtual ~MQuasiElastic();
31:
32:   void post_Constructor();
33:
34:   double operator()(const std::vector<double> &randvec, AuxIntData &aux) {
35:     return EventWeight(randvec, aux);
36:   }
37:   double EventWeight(const std::vector<double> &randvec, AuxIntData &aux);
38:   bool   EventRecord(HepMC3::GenEvent &evt);
39:   void   PrintInit(bool silent) const;
40:
41:  private:
42:   void Initialize();
43:   bool LoopKinematics(const std::vector<double> &p1p, const std::vector<double> &p2p);
44:   bool FiducialCuts() const;
45:
46:   // Non-Diffractive
47:   std::complex<double> PolySoft(const std::vector<double> &randvec);
48:
49:   // 2/3-Dim phase space, 2->2 quasielastic
50:   bool   B3RandomKin(const std::vector<double> &randvec);
51:   bool   B3BuildKin(double s3, double s4, double t);
52:   bool   B3GetLorentzScalars();
53:   double B3IntegralVolume() const;
54:   double B3PhaseSpaceWeight() const;
55:
56:   // Event by event integration boundaries
57:   double t_max         = 0.0;
58:   double t_min         = 0.0;
59:   double DD_M2_max     = 0.0;
60:   double log_DD_M2_max = 0.0;
61:
62:   // Multipomeron weight table
63:   std::vector<double> MAXPOMW;
64: };
65:
66: }  // namespace gra
67:
68: #endif
```

./include/Graniitti/MVEGAS.h 1/3

```
 1: // VEGAS integrator class grid routines
 2: //
 3: // Future step:
 4: // Factorise all VEGAS functions out from MGraniitti to here (use function pointers etc.)
 5: //
 6: // (c) 2017-2020 Mikael Mieskolainen
 7: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
 8:
 9: #ifndef MVEGAS_H
10: #define MVEGAS_H
11:
12: #include <vector>
13:
14:
15: namespace gra {
16:
17: // Vegas MC default parameters
18: struct VEGASPARAM {
19:   unsigned int BINS   = 128;  // Maximum number of bins per dimension (EVEN NUMBER!)
20:   double       LAMBDA = 1.5;  // Regularization parameter
21:
22:   // Initialization
23:   unsigned int NCALL     = 20000;  // Number of calls per iteration
24:   unsigned int ITER      = 15;     // Number of iterations
25:   double       CHI2MAX   = 10.0;   // Maximum chi2 in initialization
26:   double       PRECISION = 0.01;   // Maximum relative error of cross section integral
27:   int          DEBUG     = -1;     // Debug mode
28:
29:   // User cannot set these
30:   unsigned int MAXFDIM = 100;      // Maximum integral dimension
31:   double       EPS     = 1.0e-30;  // Epsilon parameter
32: };
33:
34:
35: // Vegas MC adaptation data
36: struct VEGASData {
37:   // VEGAS initialization function
38:   void Init(unsigned int init, const VEGASPARAM &param) {
39:     // First initialization: Create the grid and initial data
40:     if (init == 0) {
41:       ClearAll(param);
42:       InitGridDependent(param);
43:     }
44:     // Use the previous grid but NOT integral data
45:     else if (init == 1) {
46:       ClearIntegral();
47:       InitGridDependent(param);
48:     }
49:     // Use the previous grid and its integral data
50:     else if (init == 2) {
51:       InitGridDependent(param);
52:     } else {
53:       throw std::invalid_argument("VEGASData::Init: Unknown init parameter = " +
54:                                   std::to_string(init));
55:     }
56:   }
57:
58:   // Create number of calls per thread, they need to sum to calls
59:   std::vector<unsigned int> GetLocalCalls(int calls, int N_threads) {
60:     std::vector<unsigned int> localcalls(N_threads, 0.0);
61:     int                       sum = 0;
62:     for (int k = 0; k < N_threads; ++k) {
63:       localcalls[k] = std::floor(calls / N_threads);
64:       sum += localcalls[k];
65:     }
66:     // Add remainder to the thread number[0]
67:     localcalls[0] += calls - sum;
68:     return localcalls;
69:   }
70:
71:   // Initialize
72:   void InitGridDependent(const VEGASPARAM &param) {
73:     // Create grid spacing
74:     for (std::size_t j = 0; j < FDIM; ++j) { dxvec[j] = region[j + FDIM] - region[j]; }
75:
76:     // If binning parameter changed from previous call
77:     if (param.BINS != BINS_prev) {
78:       for (std::size_t i = 0; i < std::max(param.BINS, BINS_prev); ++i) { rvec[i] = 1.0; }
79:       for (std::size_t j = 0; j < FDIM; ++j) {
80:         Rebin(BINS_prev / static_cast<double>(param.BINS), j, param);
81:       }
82:       BINS_prev = param.BINS;
83:     }
```

./include/Graniitti/MVEGAS.h 2/3

```
84:   }
85:
86:   // VEGAS rebinning function (algorithm adapted from Numerical Recipes)
87:   void Rebin(double ac, unsigned int j, const VEGASPARAM &param) {
88:     unsigned int k  = 0;
89:     double       dr = 0.0;
90:     double       zn = 0.0;
91:     double       zo = 0.0;
92:
93:     for (std::size_t i = 0; i < param.BINS - 1; ++i) {
94:       while (ac > dr) { dr += rvec[(++k) - 1]; }
95:       if (k > 1) { zo = xmat[k - 2][j]; }
96:       zn = xmat[k - 1][j];
97:       dr -= ac;
98:       xcache[i] = zn - (zn - zo) * dr / rvec[k - 1];
99:     }
100:
101:    for (std::size_t i = 0; i < param.BINS - 1; ++i) { xmat[i][j] = xcache[i]; }
102:
103:    xmat[param.BINS - 1][j] = 1.0;
104:  }
105:
106:  // VEGAS grid optimizing function (algorithm adapted from Numerical Recipes)
107:  void OptimizeGrid(const VEGASPARAM &param) {
108:    double zo = 0;
109:    double zn = 0;
110:    double ac = 0;
111:
112:    for (std::size_t j = 0; j < FDIM; ++j) {
113:      zo          = f2mat[0][j];
114:      zn          = f2mat[1][j];
115:      f2mat[0][j] = (zo + zn) / 2.0;
116:      dcache[j]   = f2mat[0][j];
117:
118:      for (std::size_t i = 2; i < param.BINS; ++i) {
119:        ac              = zo + zn;
120:        zo              = zn;
121:        zn              = f2mat[i][j];
122:        f2mat[i - 1][j] = (ac + zn) / 3.0;
123:        dcache[j] += f2mat[i - 1][j];
124:      }
125:      f2mat[param.BINS - 1][j] = (zo + zn) / 2.0;
126:      dcache[j] += f2mat[param.BINS - 1][j];
127:    }
128:
129:    for (std::size_t j = 0; j < FDIM; ++j) {
130:      ac = 0.0;
131:      for (std::size_t i = 0; i < param.BINS; ++i) {
132:        f2mat[i][j] = (f2mat[i][j] < param.EPS) ? param.EPS : f2mat[i][j];
133:        rvec[i]     = std::pow((1.0 - f2mat[i][j] / dcache[j]) /
134:                               (std::log(dcache[j]) - std::log(f2mat[i][j]) + param.EPS),
135:                               param.LAMBDA);
136:        // Floating point protection (integrand close to 0)
137:        if (std::isnan(rvec[i]) || std::isinf(rvec[i])) { rvec[i] = param.EPS; }
138:        ac += rvec[i];
139:      }
140:      Rebin(ac / param.BINS, j, param);
141:    }
142:  }
143:
144:  // Initialize sampling region [0,1] x [0,1] x ... x [0,1]
145:  void InitRegion(unsigned int fdim) {
146:    FDIM = fdim;
147:    region.resize(2 * FDIM, 0.0);
148:
149:    // Lower bound [zero]
150:    for (std::size_t i = 0; i < FDIM; ++i) { region[i] = 0.0; }
151:    // Upper boundÄ [one]
152:    for (std::size_t i = FDIM; i < 2 * FDIM; ++i) { region[i] = 1.0; }
153:  }
154:
155:  // Clear integrated data
156:  void ClearIntegral() {
157:    sumdata = 0.0;
158:    sumchi2 = 0.0;
159:    sweight = 0.0;
160:  }
161:
162:  // Full init
163:  void ClearAll(const VEGASPARAM &param) {
164:    // Vectors [MAXFDIM]
165:    dcache = std::vector<double>(param.MAXFDIM, 0.0);
166:    dxvec  = std::vector<double>(param.MAXFDIM, 0.0);
```

./include/Graniitti/MVEGAS.h 3/3

```
167:
168:    // Vectors [BINS]
169:    rvec   = std::vector<double>(param.BINS, 0.0);
170:    xcache = std::vector<double>(param.BINS, 0.0);
171:
172:    // Matrices [BINS x MAXFDIM]
173:    fmat  = std::vector<std::vector<double>>(param.BINS, std::vector<double>(param.MAXFDIM, 0.0));
174:    f2mat = std::vector<std::vector<double>>(param.BINS, std::vector<double>(param.MAXFDIM, 0.0));
175:    xmat  = std::vector<std::vector<double>>(param.BINS, std::vector<double>(param.MAXFDIM, 0.0));
176:
177:    // Init with 1!
178:    for (std::size_t j = 0; j < FDIM; ++j) { xmat[0][j] = 1.0; }
179:
180:    // VEGAS integraldata
181:    sumdata = 0.0;
182:    sumchi2 = 0.0;
183:    sweight = 0.0;
184:
185:    // Previous binning
186:    BINS_prev = 1;
187:  }
188:
189:  // VEGAS scalars
190:  unsigned int BINS_prev = 0;
191:  unsigned int FDIM      = 0;
192:
193:  double fsum  = 0.0;
194:  double f2sum = 0.0;
195:
196:  double sumdata = 0.0;
197:  double sumchi2 = 0.0;
198:  double sweight = 0.0;
199:
200:  // Vectors
201:  std::vector<double> region;
202:
203:  // Vectors
204:  std::vector<double> dcache;
205:  std::vector<double> dxvec;
206:
207:  // Vectors
208:  std::vector<double> rvec;
209:  std::vector<double> xcache;
210:
211:  // Matrices
212:  std::vector<std::vector<double>> fmat;
213:  std::vector<std::vector<double>> f2mat;
214:  std::vector<std::vector<double>> xmat;
215:
216: };  // struct VEGASData
217:
218: }  // namespace gra
219:
220: #endif
```

./include/Graniitti/MFlux.h 1/1

```
// Photon and other fluxes
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

#ifndef MFLUX_H
#define MFLUX_H

// C++
#include <complex>
#include <random>
#include <vector>

// Own
#include "Graniitti/MForm.h"
#include "Graniitti/MGlobals.h"
#include "Graniitti/MKinematics.h"

namespace gra {
namespace flux {


double ApplyktEPAfluxes(double amp2, gra::LORENTZSCALAR& lts);
double ApplyOZfluxes(double amp2, gra::LORENTZSCALAR& lts);
double ApplyLUXfluxes(double amp2, gra::LORENTZSCALAR& lts);

}  // namespace flux
}  // namespace gra


#endif
```

./include/Graniitti/MCW.h 1/2

```
// Monte Carlo Weight Objects [HEADER ONLY file]
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

#ifndef MCW_H
#define MCW_H

// C++
#include <complex>
#include <random>
#include <valarray>
#include <vector>

#include "Graniitti/MMath.h"


namespace gra {
namespace kinematics {

// A simple Monte Carlo weight (container)
class MCW {
 public:
  MCW() {
    W  = 0.0;
    W2 = 0.0;
    N  = 0.0;
  }
  MCW(double w, double w2, double n) {
    W  = w;
    W2 = w2;
    N  = n;
  }
  // + operator
  MCW operator+(const MCW &obj) {
    MCW res;
    res.W  = W + obj.W;
    res.W2 = W2 + obj.W2;
    res.N  = N + obj.N;
    return res;
  }
  // += operator
  MCW &operator+=(const MCW &obj) {
    this->W += obj.W;
    this->W2 += obj.W2;
    this->N += obj.N;
    return *this;  // return by reference
  }
  // * operator (does scale W and W^2 but not N)
  MCW operator*(double scale) { return MCW(GetW() * scale, GetW2() * scale * scale, GetN()); }

  // Push new weight
  void Push(double w) {
    W += w;
    W2 += w * w;
    N += 1.0;
  }
  // MC estimate of the integral
  double Integral() const { return (N > 0.0) ? W / N : 0.0; }
  // MC estimate of the integral error squared (standard error of the mean)
  double IntegralError2() const {
    if (N > 0.0) {
      return (W2 / N - gra::math::pow2(W / N)) / N;
    } else {
      return 0.0;
    }
  }
  double IntegralError() const { return gra::math::msqrt(IntegralError2()); }
  double GetW() const { return W; }
  double GetW2() const { return W2; }
  double GetN() const { return N; }
  void   SetW(double w) { W = w; }
  void   SetW2(double w2) { W2 = w2; }
  void   SetN(double n) { N = n; }

 private:
  double W;   // sum of weights
  double W2;  // sum of weights^2
  double N;   // trials
};


// A weighted sum of MCW container integral values (use with VEGAS, for example)
```

./include/Graniitti/MCW.h 2/2

```
class MCWSUM {
 public:
  MCWSUM() {}

  void Add(const MCW &x, double weight) {
    wsum += weight;
    wintsum += weight * x.Integral();
    error2sum += (weight * weight) * x.IntegralError2();
  }
  double Integral() const {
    if (wsum > 0.0) {
      return wintsum / wsum;
    } else {
      return 0.0;
    }
  }
  double IntegralError2() const {
    if (wsum > 0.0) {
      return error2sum / gra::math::pow2(wsum);
    } else {
      return 0.0;
    }
  }
  double IntegralError() const { return gra::math::msqrt(IntegralError2()); }

 private:
  // \sum_i weight_i
  double wsum = 0.0;

  // \sum_i weight_i integral_i
  double wintsum = 0.0;

  // \sum_i weight_i^2 error_i^2
  double error2sum = 0.0;
};

}  // namespace kinematics
}  // namespace gra

#endif
```

./include/Graniitti/MDurham.h 1/2

```
// "Durham QCD" Processes and Amplitudes
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

#ifndef MDURHAM_H
#define MDURHAM_H

// C++
#include <complex>
#include <random>
#include <vector>

// Own
#include "Graniitti/M4Vec.h"
#include "Graniitti/MAmplitudes.h"
#include "Graniitti/MAux.h"
#include "Graniitti/MGlobals.h"
#include "Graniitti/MKinematics.h"
#include "Graniitti/MSudakov.h"

namespace gra {

// Durham loop integral discretization technical parameters
struct MDurhamParam {
  unsigned int N_qt  = 0;  // (> 30)
  unsigned int N_phi = 0;  // (> 10)

  double qt2_MIN = 0;  // Loop momentum qt^2 minimum (GeV^2)
  double qt2_MAX = 0;  // Loop momentum qt^2 maximum (GeV^2)

  std::string PDF_scale    = "MIN";  // Scheme
  double      alphas_scale = 4.0;    // PDF factorization scale
  double      MAXCOS       = 0.9;    // Meson amplitude |cos(theta*)| < MAXCOS

  // THESE ARE CALCULATED FROM ABOVE
  double qt_MIN      = 0;
  double qt_MAX      = 0;
  double qt_STEP     = 0;
  double phi_STEP    = 0;
  bool   initialized = false;

  // Read parameters from file
  void ReadParameters(const std::string &modelfile) {
    using json            = nlohmann::json;
    const std::string data = gra::aux::GetInputData(modelfile);
    json              j;

    try {
      j = json::parse(data);

      // JSON block identifier
      const std::string XID = "PARAM_DURHAMQCD";
      N_qt                  = j.at(XID).at("N_qt");
      N_phi                 = j.at(XID).at("N_phi");
      qt2_MIN               = j.at(XID).at("qt2_MIN");
      qt2_MAX               = j.at(XID).at("qt2_MAX");
      PDF_scale             = j.at(XID).at("PDF_scale");
      alphas_scale          = j.at(XID).at("alphas_scale");
      MAXCOS                = j.at(XID).at("MAXCOS");

      // Now calculate rest
      qt_MIN = math::msqrt(qt2_MIN);
      qt_MAX = math::msqrt(qt2_MAX);

      qt_STEP  = (qt_MIN - qt_MAX) / N_qt;
      phi_STEP = (2.0 * math::PI) / N_phi;

      initialized = true;
    } catch (...) {
      std::string str = "MDurhamParam::ReadParameters: Error parsing " + modelfile +
                        " (Check for extra/missing commas)";
      throw std::invalid_argument(str);
    }
  }
};

class MDurham : public MAmplitudes {
 public:
  MDurham(gra::LORENTZSCALAR &lts, const std::string &modelfile);
  ~MDurham() {}
```

```
84:   double      DurhamQCD(gra::LORENTZSCALAR &lts, const std::string &process);
85:   double      DQtloop(gra::LORENTZSCALAR &lts, std::vector<std::vector<std::complex<double>>> Amp);
86:   inline void DScaleChoise(double qt2, double q1_2, double q2_2, double &Q1_2_scale,
87:                            double &Q2_2_scale) const;
88:
89:   inline void Dgg2chic0(const gra::LORENTZSCALAR &                          lts,
90:                         std::vector<std::vector<std::complex<double>>> &Amp,
91:                         const std::vector<double> &qt1, const std::vector<double> &qt2) const;
92:
93:   inline void DHelicity(const std::vector<double> &q1, const std::vector<double> &q2,
94:                         std::vector<std::complex<double>> &J2P) const;
95:
96:   inline std::complex<double> DHelProj(const std::vector<std::complex<double>> &A,
97:                                        const std::vector<std::complex<double>> &J2P) const;
98:
99:   void Dgg2gg(const gra::LORENTZSCALAR &lts, std::vector<std::vector<std::complex<double>>> &Amp);
100:
101:  void Dgg2qqbar(const gra::LORENTZSCALAR &                          lts,
102:                 std::vector<std::vector<std::complex<double>>> &Amp);
103:
104:  void                Dgg2MMbar(const gra::LORENTZSCALAR &                          lts,
105:                                std::vector<std::vector<std::complex<double>>> &Amp);
106:  double              phi_CZ(double x, double fM) const;
107:  std::vector<double> EvalPhi(int N, int pdg) const;
108:
109:  double Asum = 0.0;
110:  double Nsum = 0.0;
111:
112: private:
113:  // Parameters
114:  MDurhamParam Param;
115: };
116:
117: }  // namespace gra
118:
119: #endif
```



```
1: // Abstract process class
2: //
3: // (c) 2017-2020 Mikael Mieskolainen
4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
5:
6: #ifndef MPROCESS_H
7: #define MPROCESS_H
8:
9: // C++
10: #include <complex>
11: #include <random>
12: #include <vector>
13:
14: // HepMC33
15: #include "HepMC3/FourVector.h"
16: #include "HepMC3/GenEvent.h"
17: #include "HepMC3/GenParticle.h"
18:
19: // Own
20: #include "Graniitti/M4Vec.h"
21: #include "Graniitti/MAux.h"
22: #include "Graniitti/MEikonal.h"
23: #include "Graniitti/MGlobals.h"
24: #include "Graniitti/MH1.h"
25: #include "Graniitti/MH2.h"
26: #include "Graniitti/MHELMatrix.h"
27: #include "Graniitti/MKinematics.h"
28: #include "Graniitti/MPDG.h"
29: #include "Graniitti/MRandom.h"
30: #include "Graniitti/MSubProc.h"
31: #include "Graniitti/MSudakov.h"
32: #include "Graniitti/MUserHistograms.h"
33:
34: namespace gra {
35:
36: // Event-by-event auxiliary data for integration
37: struct AuxIntData {
38:   // Aux weight
39:   double vegasweight  = 1.0;
40:   bool   burn_in_mode = false;
41:
42:   // Event-by-event assertations (init all with true!)
43:   bool amplitude_ok  = true;
44:   bool kinematics_ok = true;
45:   bool fidcuts_ok    = true;
46:   bool vetocuts_ok   = true;
47:
48:   // Forced acceptance of the event
49:   bool forced_accept = false;
50:
51:   bool Valid() const { return kinematics_ok && fidcuts_ok && vetocuts_ok; }
52: };
53:
54: // Multipomeron kinematics
55: struct MPI {
56:   M4Vec p1i;
57:   M4Vec p2i;
58:
59:   M4Vec q1;
60:   M4Vec q2;
61:
62:   M4Vec k;
63:
64:   M4Vec p1f;
65:   M4Vec p2f;
66:
67:   M4Vec p3;
68:   M4Vec p4;
69: };
70:
71: // Abstract process class
72: class MProcess : public MUserHistograms {
73:  public:
74:   // Of polymorphic type
75:   virtual ~MProcess() {}  // MUST HAVE IT HERE as virtual
76:
77:   // Pure virtual functions, without definitions here
78:   // WITHOUT =0, undefined reference to vtable for MProcess will occur in
79:   // compilation
80:   virtual void post_Constructor()           = 0;
81:   virtual void PrintInit(bool silent) const = 0;
82:
83:   virtual double operator()(const std::vector<double> &randvec, AuxIntData &aux)  = 0;
```



```
84:   virtual double EventWeight(const std::vector<double> &randvec, AuxIntData &aux) = 0;
85:   virtual bool   EventRecord(HepMC3::GenEvent &evt)                               = 0;
86:
87:   // Set central system decay structure
88:   void      SetDecayMode(std::string str);
89:   void      SetupBranching();
90:   void      ProcessHelicityTree(MDecayBranch &branch);
91:   HELMatrix ProcessHelicityDecay(const MParticle &p, const std::vector<MParticle> &daughter) const;
92:
93:   // Set initial state
94:   void SetInitialState(const std::vector<std::string> &beam, const std::vector<double> &energy);
95:
96:   // Set beam energies
97:   void SetBeamEnergies(double E1, double E2);
98:
99:   // ISOLATE phase space in <F> class processes
100:  void SetISOLATE(bool in) { lts.PS_active = !in; }
101:  bool GetISOLATE() { return !lts.PS_active; }
102:  void SetFLATMASS2(bool in) {
103:    aux::PrintNotice();
104:    std::cout << rang::fg::red
105:              << "MProcess::SetFLATMASS2: Set flat in mass^2 "
106:              "sampling in decay trees: "
107:              << rang::fg::reset << std::endl;
108:    FLATMASS2      = in;
109:    FLATMASS2_user = true;  // user has tagged it
110:  }
111:  bool GetFLATMASS2() { return FLATMASS2; }
112:  void SetOFFSHELL(double in) {
113:    aux::PrintNotice();
114:    std::cout << rang::fg::red
115:              << "MProcess::SetOFFSHELL: Set number of "
116:              "decay widths in decay trees: "
117:              << in << rang::fg::reset << std::endl;
118:    OFFSHELL      = in;
119:    OFFSHELL_user = true;  // user has tagged it
120:  }
121:  double GetOFFSHELL() { return OFFSHELL; }
122:
123:  // Set generation spin correlations
124:  void SetSPINGEN(const bool SPINGEN) {
125:    std::cout << rang::fg::red << "MProcess::SetSPINGEN: Set generation 2->1 spin correlations: "
126:              << (SPINGEN ? "true" : "false") << rang::fg::reset << std::endl;
127:    for (const auto &x : lts.RESONANCES) { lts.RESONANCES[x.first].SPINGEN = SPINGEN; }
128:  }
129:  // Set decay spin correlations
130:  void SetSPINDEC(const bool SPINDEC) {
131:    std::cout << rang::fg::red << "MProcess::SetSPINDEC: Set decay 1->2 spin correlations: "
132:              << (SPINDEC ? "true" : "false") << rang::fg::reset << std::endl;
133:    for (const auto &x : lts.RESONANCES) { lts.RESONANCES[x.first].SPINDEC = SPINDEC; }
134:  }
135:  // Set common Lorentz frame for all resonances
136:  void SetFRAME(const std::string &FRAME) {
137:    std::cout << rang::fg::red
138:              << "MProcess::SetFRAME: Set common Lorentz "
139:              "frame for the resonance amplitudes: "
140:              << FRAME << rang::fg::reset << std::endl;
141:    for (const auto &x : lts.RESONANCES) { lts.RESONANCES[x.first].FRAME = FRAME; }
142:  }
143:
144:  // Set maximum sliding pomeron helicity for all resonances
145:  void SetJMAX(const int &JMAX) {
146:    std::cout << rang::fg::red
147:              << "MProcess::SetJMAX: Set common maximum Pomeron helicity "
148:              "for all the resonances: "
149:              << JMAX << rang::fg::reset << std::endl;
150:    for (const auto &x : lts.RESONANCES) { lts.RESONANCES[x.first].JMAX = JMAX; }
151:  }
152:
153:  // Get initial state
154:  std::vector<gra::MParticle> GetInitialState() {
155:    std::vector<gra::MParticle> beams = {lts.beam1, lts.beam2};
156:    return beams;
157:  }
158:
159:  // Phase space dimension
160:  unsigned int GetdLIPSDim() { return ProcPtr.LIPSDIM; }
161:
162:  // Pomeron loop screening
163:  void SetScreening(bool value) { SCREENING = value; }
164:  bool GetScreening() { return SCREENING; }
165:  // Set/Get input eikonal
166:  void     SetEikonal(const MEikonal &in) { Eikonal = in; }
```



```
167:  MEikonal GetEikonal() const { return Eikonal; }
168:
169:  // Set LHAPDFSET name
170:  void SetLHAPDF(const std::string &in) {
171:    std::cout << "MProcess::SetLHAPDF: " << in << std::endl;
172:    lts.LHAPDFSET = in;
173:  }
174:
175:  // Set cuts
176:  void SetGenCuts(const gra::GENCUT &in) { gcuts = in; }
177:  void SetFidCuts(const gra::FIDCUT &in) { fcuts = in; }
178:  void SetUserCuts(int in) { USERCUTS = in; }
179:  void SetVetoCuts(const gra::VETOCUT &in) { vetocuts = in; }
180:
181:  double GetMandelstam_s() const { return lts.s; }
182:
183:  // Set proton excitation to low-mass N*
184:  void SetExcitation(int in) {
185:    if (in > 2 || in < 0) {
186:      std::string str =
187:          "MProcess::SetExcitation: Not valid input "
188:          "(0,1,2) : " +
189:          std::to_string(in) + " !";
190:      throw std::invalid_argument(str);
191:    }
192:    EXCITATION = in;
193:
194:    if (EXCITATION > 0) {
195:      aux::PrintWarning();
196:      std::cout << rang::fg::red
197:                << "MProcess::SetExcitation: Proton "
198:                   "excitation is under construction / some processes contain only kinematic part: "
199:                << in << rang::fg::reset << std::endl;
200:    }
201:  }
202:
203:  // Set flat matrix element mode
204:  void SetFLATAMP(int in) {
205:    if (in > 0) {
206:      aux::PrintNotice();
207:      std::cout << rang::fg::red << "MProcess::SetFLATAMP: Flat matrix element FLATAMP: " << in
208:                << rang::fg::reset << std::endl;
209:    }
210:    FLATAMP = in;
211:  }
212:
213:  // pp invariant Moller flux (high energy limit)
214:  double MollerFlux() { return 2.0 * lts.s; }
215:
216:  // Flat amplitudes (for DEBUG)
217:  double GetFlatAmp2(const gra::LORENTZSCALAR &lts) const;
218:
219:  // Set input resonances
220:  void SetResonances(const std::map<std::string, gra::PARAM_RES> &in) { lts.RESONANCES = in; }
221:
222:  // Get input resonances
223:  std::map<std::string, gra::PARAM_RES> GetResonances() const { return lts.RESONANCES; }
224:
225:  // Eikonal (screening) functions
226:  MEikonal Eikonal;
227:
228:  // Lets keep these public for easy access
229:  gra::LORENTZSCALAR lts;  // Lorentz scalars and others for kinematics
230:
231:  // Cut structures
232:  gra::GENCUT  gcuts;     // Generator sampling cuts (phase space boundaries)
233:  gra::FIDCUT  fcuts;     // Fiducial cuts (phase space boundaries)
234:  gra::VETOCUT vetocuts;  // Veto cuts
235:
236:  void PrintDecayTree(const gra::MDecayBranch &branch) const;
237:
238:  void CalculatePhaseSpace(const gra::MDecayBranch &branch, double &product, double &product2pi,
239:                           double &volume, int &N_final) const;
240:  void PrintPhaseSpace(const gra::MDecayBranch &branch, double &product, double &product2pi,
241:                       int &N_final) const;
242:
243:  // Random numbers (keep it public for seeding)
244:  MRandom random;
245:
246:  // Subprocess (amplitudes)
247:  MSubProc ProcPtr;
248:
249: protected:
```

./include/Graniitti/MProcess.h 4/5

```
250:   // Copy and assignment made private
251:   // MProcess(const MProcess& other);
252:   // MProcess& operator=(const MProcess& rhs);
253:
254:   // Internal virtual functions, without definitions here
255:   virtual void Initialize()                                                                    = 0;
256:   virtual bool LoopKinematics(const std::vector<double> &p1p, const std::vector<double> &p2p) = 0;
257:   virtual bool FiducialCuts() const                                                            = 0;
258:
259:   // Cascade phase-space factor
260:   double CascadePS() const;
261:
262:   // Amplitude squared
263:   double GetAmp2();
264:
265:   // Eikonal screening loop
266:   double S3ScreenedAmp2();
267:
268:   // First print
269:   void PrintSetup() const;
270:
271:   // Setup process
272:   void SetProcess(std::string &process, const std::vector<aux::OneCMD> &syntax);
273:
274:   // QFT symmetry factor
275:   void CalculateSymmetryFactor();
276:
277:   // ------------------------------------------------------
278:   // Recursive function to treat decay trees
279:
280:   void SaveBranch(HepMC3::GenEvent &evt, const gra::MDecayBranch &branch,
281:                   const HepMC3::GenParticlePtr &pX);
282:   bool CommonRecord(HepMC3::GenEvent &evt);
283:   bool VetoCuts() const;
284:   bool CommonCuts() const;
285:   void FindDecayCuts(const gra::MDecayBranch &branch, bool &ok) const;
286:   void FindVetoCuts(const gra::MDecayBranch &branch, bool &ok) const;
287:   bool ConstructDecayKinematics(gra::MDecayBranch &branch);
288:   void WriteDecayKinematics(const gra::MDecayBranch &branch, const HepMC3::GenParticlePtr &mother,
289:                             HepMC3::GenEvent &evt);
290:   void PrintFiducialCuts() const;
291:
292:   void   GetOffShellMass(const gra::MDecayBranch &branch, double &mass);
293:   void   SetTechnicalBoundaries(gra::GENCUT &gcuts, unsigned int EXCITATION);
294:   double ForwardVolume() const;
295:
296:   // Lorentz scalars
297:   bool GetLorentzScalars(unsigned int Nf);
298:
299:   void SampleForwardMasses(std::vector<double> &mvec, const std::vector<double> &randvec);
300:
301:   // ------------------------------------------------------
302:   // System fragmentation
303:
304:   bool ExciteNstar(const M4Vec &nstar, gra::MDecayBranch &forward, const MParticle &pbeam);
305:   bool ExciteContinuum(const M4Vec &nstar, gra::MDecayBranch &forward, double Q2_scale, int B_sum,
306:                        int Q_sum, const std::string &pt_distribution = "powexp");
307:   void BranchForwardSystem(const std::vector<M4Vec> &p4, const std::vector<MParticle> &p,
308:                            const M4Vec &nstar, gra::MDecayBranch &forward);
309:   bool CEPForwardFragment();
310:
311:   // ------------------------------------------------------
312:
313:   void ParseCMD(const std::string &str, std::string &first, std::string &second,
314:                 std::string &third) const;
315:
316:   // Check std::nan/std::inf
317:   bool CheckInfNan(double &W) {
318:     if (std::isnan(W)) {
319:       ++N_nan;
320:       W = 0;
321:       return false;
322:     } else if (std::isinf(W)) {
323:       ++N_inf;
324:       W = 0;
325:       return false;
326:     }
327:     if (N_nan > 50) {
328:       throw std::invalid_argument(
329:           "MProcess::CheckInfNan: Too many NaN weights "
330:           "- Check model parameters "
331:           "and cuts!");
332:     }
```

./include/Graniitti/MProcess.h 5/5

```
333:     if (N_inf > 50) {
334:       throw std::invalid_argument(
335:           "MProcess::CheckInfNan: Too many Inf weights "
336:           "- Check model parameters "
337:           "and cuts!");
338:     }
339:     return true;
340:   }
341:   unsigned int N_inf = 0;
342:   unsigned int N_nan = 0;
343:
344:   // Cross section statistical 1/S symmetry factor (for identical final states)
345:   double S_factor = 0.0;
346:
347:   // ------------------------------------------------------------------
348:   // Steering parameters
349:
350:   std::string PROCESS;             // Process identifier string
351:   std::string CID;                 // Phase space sampler identifier such as "F" or "C"
352:   std::string DECAYMODE;           // Decaymode identifier string
353:   bool        SCREENING  = false;  // Pomeron loop on/off
354:   int         EXCITATION = 0;      // Forward proton excitation (0 = off, 1 = single, 2 = double)
355:   int         USERCUTS   = 0;      // User custom cuts identifier
356:   int         FLATAMP    = 0;      // Flat matrix element mode
357:
358:   // ------------------------------------------------------------------
359:   // Phase-space control
360:
361:   bool   FLATMASS2      = false;  // Flat in M^2 instead of Breit-Wigner sampling
362:   bool   FLATMASS2_user = false;
363:   double OFFSHELL       = 5;  // How many full widths to sample particles in cascades
364:   bool   OFFSHELL_user  = false;
365:
366:   // Forward excitation minimum/maximum M^2 boundaries
367:   double M2_f_min     = 0.0;
368:   double M2_f_max     = 0.0;
369:   double log_M2_f_min = 0.0;
370:   double log_M2_f_max = 0.0;
371:
372:   static constexpr double ZERO_EPS = 1e-12;  // To use with log(0+ZERO_EPS)
373:   // ------------------------------------------------------------------
374:
375:   // Non-Diffractive
376:   std::vector<MPI> etree;
377:   double          bt = 0.0;
378: };
379:
380: }  // namespace gra
381:
382: #endif
```

./include/Graniitti/MUserCuts.h 1/1

```
1: // Custom user defined cuts
2: //
3: // (c) 2017-2020 Mikael Mieskolainen
4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
5:
6: #ifndef MUSERCUTS_H
7: #define MUSERCUTS_H
8:
9: // C++
10: #include <vector>
11:
12: // Own
13: #include "Graniitti/MKinematics.h"
14:
15: namespace gra {
16: // User cuts (return false for events not passing the cuts)
17: bool UserCut(int id, const gra::LORENTZSCALAR &lts);
18:
19: }  // namespace gra
20:
21: #endif
```

./include/Graniitti/MPDG.h 1/2

```
1: // PDG class
2: //
3: // (c) 2017-2019 Mikael Mieskolainen
4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
5:
6: #ifndef MPDG_H
7: #define MPDG_H
8:
9: // C++
10: #include <complex>
11: #include <fstream>
12: #include <map>
13: #include <random>
14: #include <vector>
15:
16: // Own
17: #include "Graniitti/MAux.h"
18: #include "Graniitti/MParticle.h"
19:
20: namespace gra {
21: namespace PDG {
22:
23: // Common PDG ids, http://pdg.lbl.gov/2007/reviews/montecarlorpp.pdf
24: // For MC internal, use 81-100
25: constexpr int PDG_p     = 2212;  // Proton
26: constexpr int PDG_n     = 2112;  // ...
27: constexpr int PDG_pip   = 211;
28: constexpr int PDG_pim   = -211;
29: constexpr int PDG_pi0   = 111;
30: constexpr int PDG_Kp    = 321;
31: constexpr int PDG_Km    = -321;
32: constexpr int PDG_gamma = 22;
33: constexpr int PDG_gluon = 21;
34: constexpr int PDG_muon  = 13;
35: constexpr int PDG_rho0  = 113;
36:
37: constexpr int PDG_pomeron    = 990;
38: constexpr int PDG_reggeon    = 110;
39: constexpr int PDG_monopole   = 992;   // spin-1/2 monopole
40: constexpr int PDG_MSTAR     = 90000;  // Meson  resonance
41: constexpr int PDG_NSTAR     = 90210;  // Baryon resonance
42: constexpr int PDG_system    = 90;     // "Container" / "Jet" / "System"
43: constexpr int PDG_fragment  = 91;     // Proton fragment
44: constexpr int PDG_propagator = 99;    // Generic propagator
45:
46: // HepPMC STABLE definitions conventions
47: constexpr int PDG_BEAM       = 4;   // Beam particles
48: constexpr int PDG_STABLE     = 1;   // Final state "stable" particles
49: constexpr int PDG_DECAY      = 2;   // Such as pi0
50: constexpr int PDG_INTERMEDIATE = 81;  // Such as gamma*, pomeron etc.
51:
52:
53: // For CPU efficiency [GeV]
54: constexpr double mp   = 0.938272081;
55: constexpr double mK   = 0.493677;
56: constexpr double mpi  = 0.13957018;
57: constexpr double mpi0 = 0.13497660;
58:
59: // [hbar] = [M][L]^(2)[T]^(-1) and [c] = [L][T]^(-1)
60: // set (hbar = c = 1)
61: // then [M] = [L]^(-1) = T^(-1)
62:
63: // http://pdg.lbl.gov/2018/reviews/rpp2018-rev-phys-constants.pdf
64: // Basic constants
65: constexpr double c    = 2.99792458E8;      // c    = [m/s] (EXACT/DEFINITION)
66: constexpr double hbar = 6.582119514E-25;   // hbar = [GeV*s]
67: constexpr double eV   = 1.6021766208E-19;  // e    = [Joule], e^0.303 in nat.u.
68:
69: // Basic definition
70: constexpr double barn2m2 = 1E-28;  // 1 barn to [m^2]
71:
72: // Standard conversions
73: constexpr double GeV2m  = hbar * c;       // 1 GeV^(-1) to [m]
74: constexpr double GeV2fm = GeV2m * 1E15;   // 1 GeV^(-1) to [fermi] (1 fm)
75: constexpr double GeV2m2 = GeV2m * GeV2m;  // 1 GeV^(-2) to [m^2]
76: constexpr double GeV2s  = hbar;          // 1 GeV^(-1) to [s]
77:
78: constexpr double GeV2barn = GeV2m2 / barn2m2;  // 1 GeV^(-2) to barns
79: constexpr double GeV2mb  = GeV2barn * 1E3;    // 1 GeV^(-2) to mbarns
80: // constexpr double GeV2ub = GeV2barn * 1E6;  // 1 GeV^(-2) to microbarn
81: // constexpr double GeV2nb = GeV2barn * 1E9;  // 1 GeV^(-2) to nanobarn
82: // constexpr double GeV2pb = GeV2barn * 1E12; // 1 GeV^(-2) to picobarns
83: // constexpr double mb2GeV = 1.0 / GeV2mb;    // 1 mb to GeV^(-2)
```

./include/Graniitti/MPDG.h 2/2

```
  84:
  85: // More conversions
  86: // constexpr double GeV2J   = eV * 1E9; // 1 GeV^{1} to [kg*m^2/s^2]=[Joule]
  87: // constexpr double GeV2kg  = GeV2J / GeV2m2 * pow2(GeV2s); // 1 GeV^{1} to [kg]
  88: // constexpr double GeV2N   = GeV2J / GeV2m; // 1 GeV^{2} to [N]=[kg*m/s^2]
  89: // (Force)
  90: // constexpr double GeV2mom = GeV2J / GeV2m * GeV2s; // 1 GeV^{1} to [kg*m/s]
  91: // (Momentum)
  92:
  93: // cross section: [value] x [GeV^{-2}] = [value] x [hbar x c]^2
  94: // decay rate:    [value] x [GeV] = [value] / hbar
  95: // length:        [value] x [GeV^{-1}] = [value] x [hbar x c]
  96:
  97: // Meson decay constants (PDG) [pi0, pi+, K+, K0]
  98: static const std::map<int, double> fM_meson{
  99:     {111, 0.1300}, {211, 0.1307}, {321, 0.1598}, {311, 0.1598}};
 100:
 101: }  // namespace PDG
 102:
 103: class MPDG {
 104:  public:
 105:   MPDG() {}
 106:   ~MPDG() {}
 107:
 108:   void ReadParticleData(const std::string &filepath);
 109:   void TokenizeProcess(const std::string &str, int depth,
 110:                        std::vector<gra::MDecayBranch> &branches) const;
 111:   bool IsDecay(const std::string &str) const;
 112:
 113:   void                  PrintPDGTable() const;
 114:   const gra::MParticle &FindByPDG(int pdgcode) const;
 115:   const gra::MParticle &FindByPDGName(const std::string &pdgname) const;
 116:
 117:   // PDG tables
 118:   std::map<int, gra::MParticle> PDG_table;
 119: };
 120:
 121: }  // namespace gra
 122:
 123: #endif
```

./include/Graniitti/MRandom.h 1/1

```
   1: // Random number class
   2: //
   3: // (c) 2017-2020 Mikael Mieskolainen
   4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
   5:
   6: #ifndef MRANDOM_H
   7: #define MRANDOM_H
   8:
   9: // C++
  10: #include <complex>
  11: #include <random>
  12: #include <vector>
  13:
  14: namespace gra {
  15: class MRandom {
  16:  public:
  17:   // Calling constructors of member functions
  18:   MRandom() : flat(0, 1), gaussian(0, 1) {
  19:     rng.seed();  // Default initialization
  20:   }
  21:   ~MRandom() {}
  22:
  23:   // Set random number engine seed
  24:   void SetSeed(int seed) {
  25:     const int SEEDMAX = 2147483647;
  26:     if (seed > SEEDMAX) {
  27:       std::string str = "MRandom::SetSeed: Invalid input seed: " + std::to_string(seed) +
  28:                                 " > SEEDMAX = " + std::to_string(SEEDMAX);
  29:       throw std::invalid_argument(str);
  30:     }
  31:     rng.seed(seed);
  32:     RNDSEED = seed;
  33:   }
  34:
  35:   // Return current random seed
  36:   unsigned int GetSeed() const { return RNDSEED; }
  37:
  38:   // Random sampling functions
  39:   double U(double a, double b);
  40:   double G(double mu, double sigma);
  41:   double PowerRandom(double a, double b, double alpha);
  42:   double RelativisticBWRandom(double m0, double Gamma, double LIMIT = 5.0, double M_MIN = 0.0);
  43:   double CauchyRandom(double m0, double Gamma, double LIMIT = 5.0, double M_MIN = 0.0);
  44:   int    NBDRandom(double avgN, double k, int maxvalue);
  45:   int    PoissonRandom(double lambda);
  46:   double ExpRandom(double lambda);
  47:   int    LogRandom(double p, int maxvalue);
  48:   void   DirRandom(const std::vector<double> &alpha, std::vector<double> &y);
  49:
  50:   double NBDpdf(int n, double avgN, double k);
  51:   double Logpdf(int k, double p);
  52:
  53:   // For generators, check: https://nullprogram.com/blog/2017/09/21/
  54:
  55:   // 64-bit Mersenne Twister by Matsumoto and Nishimura, fast, basic
  56:   // For some (possible) sources problems, see:
  57:   // [REFERENCE: Harase, https://arxiv.org/abs/1708.06018]
  58:   // std::mt19937_64 rng;
  59:
  60:   // 48-bit RANLUX (a bit slower)
  61:   // [REFERENCE: Luscher, https://arxiv.org/abs/hep-lat/9309020]
  62:   std::ranlux48 rng;
  63:
  64:   // Distribution engines
  65:   std::uniform_real_distribution<double> flat;
  66:   std::normal_distribution<double>       gaussian;
  67:   unsigned int                           RNDSEED = 0;  // Random seed set
  68:
  69:  private:
  70:   // Nothing
  71: };
  72:
  73: }  // namespace gra
  74:
  75: #endif
```

./include/Graniitti/MEikonal.h 1/4

```
   1: // Eikonal proton density and Screening amplitude class
   2: //
   3: // (c) 2017-2020 Mikael Mieskolainen
   4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
   5:
   6: #ifndef MEIKONAL_H
   7: #define MEIKONAL_H
   8:
   9: // C++
  10: #include <complex>
  11: #include <vector>
  12:
  13: // Own
  14: #include "Graniitti/M4Vec.h"
  15: #include "Graniitti/MForm.h"
  16: #include "Graniitti/MMath.h"
  17: #include "Graniitti/MMatrix.h"
  18: #include "Graniitti/MPDG.h"
  19:
  20: // External
  21: #include "rang.hpp"
  22:
  23: namespace gra {
  24:
  25: // Numerical integration parameters
  26: struct MEikonalNumerics {
  27:   static constexpr double MinKT2    = 1E-6;
  28:   static constexpr double MaxKT2    = 25.0;
  29:   unsigned int            NumberKT2 = 0;
  30:   bool                    logKT2    = false;
  31:
  32:   static constexpr double MinBT     = 1E-6;
  33:   static constexpr double MaxBT     = 10.0 / PDG::GeV2fm;
  34:   unsigned int            NumberBT = 0;
  35:   bool                    logBT    = false;
  36:
  37:   static constexpr double       FBIntegralMinKT = 1E-9;
  38:   static constexpr double       FBIntegralMaxKT = 30.0;
  39:   static constexpr unsigned int FBIntegralN     = 10000;
  40:   static constexpr double       MinLoopKT       = 1E-4;
  41:
  42:   double MaxLoopKT = 1.75;
  43:
  44:   unsigned int NumberLoopKT  = 15;  // Number of kt steps  (default minimum)
  45:   unsigned int NumberLoopPHI = 12;  // Number of phi steps (default minimum)
  46:
  47:   // User setup (ND can be negative, to get below the default)
  48:   void SetLoopDiscretization(int ND) {
  49:     NumberLoopKT  = std::max(3, 3 * ND + (int)NumberLoopKT);
  50:     NumberLoopPHI = std::max(3, 3 * ND + (int)NumberLoopPHI);
  51:
  52:     if (ND < 0) { std::cout << rang::fg::red; }
  53:     std::cout << "MEikonalNumerics::SetLoopDiscretization: ND = " << ND << std::endl;
  54:     std::cout << "-> NumberLoopKT  = " << NumberLoopKT << std::endl;
  55:     std::cout << "-> NumberLoopPHI = " << NumberLoopPHI << std::endl;
  56:     std::cout << rang::fg::reset;
  57:   }
  58:
  59:   // Unique hash
  60:   std::string GetHashString() {
  61:     std::string str = std::to_string(MinKT2) + std::to_string(MaxKT2) + std::to_string(NumberKT2) +
  62:                       std::to_string(logKT2) + std::to_string(MinBT) + std::to_string(MaxBT) +
  63:                       std::to_string(NumberBT) + std::to_string(logBT) +
  64:                       std::to_string(FBIntegralMinKT) + std::to_string(FBIntegralMaxKT) +
  65:                       std::to_string(FBIntegralN);
  66:     return str;
  67:   }
  68:
  69:   // Read parameters from file
  70:   void ReadParameters() {
  71:     // Read and parse
  72:     using json = nlohmann::json;
  73:
  74:     const std::string inputfile = gra::aux::GetBasePath(2) + "/modeldata/" + "NUMERICS.json";
  75:     const std::string data      = gra::aux::GetInputData(inputfile);
  76:     json              j;
  77:
  78:     try {
  79:       j = json::parse(data);
  80:
  81:       // JSON block identifier
  82:       const std::string XID = "NUMERICS_EIKONAL";
  83:
```

./include/Graniitti/MEikonal.h 2/4

```
  84:       NumberKT2 = j.at(XID).at("NumberKT2");
  85:       logKT2    = j.at(XID).at("logKT2");
  86:       NumberBT  = j.at(XID).at("NumberBT");
  87:       logBT     = j.at(XID).at("logBT");
  88:
  89:     } catch (...) {
  90:       std::string str =
  91:           "ReadParameters: Error parsing " + inputfile + " (Check for extra/missing commas)";
  92:       throw std::invalid_argument(str);
  93:     }
  94:   }
  95: };
  96:
  97:
  98: // Interpolation container
  99: class IArray1D {
 100:  public:
 101:   IArray1D(){};
 102:   ~IArray1D(){};
 103:
 104:   std::string  name;
 105:   double       MIN   = 0;
 106:   double       MAX   = 0;
 107:   unsigned int N     = 0;
 108:   double       STEP  = 0;
 109:   bool         islog = false;
 110:
 111:   // Setup discretization
 112:   void Set(std::string _name, double _min, double _max, double _N, double _logarithmic) {
 113:     // At least two intervals
 114:     if (_N < 2) {
 115:       throw std::invalid_argument("IArray1D::Set: Error: N = " + std::to_string(_N) + " < 2");
 116:     }
 117:
 118:     // Check sanity
 119:     if (_min >= _max) {
 120:       throw std::invalid_argument("IArray1D::Set: Error: Variable " + _name + " MIN = " +
 121:                                   std::to_string(_min) + " >= MAX = " + std::to_string(_max));
 122:     }
 123:
 124:     // Check sanity
 125:     if (_logarithmic && _min < 1e-9) {
 126:       throw std::invalid_argument(
 127:           "IArray1D::Set: Error: Variable " + _name +
 128:           " is using logarithmic stepping with boundary MIN = " + std::to_string(_min));
 129:     }
 130:
 131:     islog = _logarithmic;
 132:
 133:     // Logarithm taken here!
 134:     MIN = islog ? std::log(_min) : _min;
 135:     MAX = islog ? std::log(_max) : _max;
 136:     N   = _N;
 137:
 138:     STEP = (MAX - MIN) / N;
 139:     name = _name;
 140:   }
 141:
 142:   // Call this last
 143:   void InitArray() {
 144:     // Note N+1 !
 145:     F = MMatrix<std::complex<double>>(N + 1, 2, 0.0);
 146:   }
 147:
 148:   // N+1 size!
 149:   MMatrix<std::complex<double>> F;
 150:
 151:   // CMS energy (needed for boundary conditions)
 152:   double sqrts = 0.0;
 153:
 154:   std::string GetHashString() const {
 155:     std::string str = std::to_string(islog) + std::to_string(MIN) + std::to_string(MAX) +
 156:                       std::to_string(N) + std::to_string(sqrts);
 157:     return str;
 158:   }
 159:
 160:   bool                 WriteArray(const std::string &filename, bool overwrite) const;
 161:   bool                 ReadArray(const std::string &filename);
 162:   std::complex<double> InterpolateID(double a) const;
 163:
 164:   static const unsigned int X = 0;
 165:   static const unsigned int Y = 1;
 166: };
```

./include/Graniitti/MEikonal.h 3/4

```
167:
168: class MEikonal {
169:  public:
170:   MEikonal();
171:   ~MEikonal();
172:
173:   // Construct outside
174:   void S3Constructor(double s_in, const std::vector<gra::MParticle> &initialstate_in,
175:                      bool onlydensity = false, int NumberBT = 0, int NumberKT2 = 0);
176:
177:   // Get eikonal and amplitude values
178:   // std::complex<double> S3DensityInterpolator(double bt);
179:   // std::complex<double> S3ScreeningInterpolator(double kt2);
180:
181:   // Initialization already done
182:   bool IsInitialized() const {
183:     if (S3INIT == true) { return true; }
184:     return false;
185:   }
186:
187:   // Get total cross sections
188:   void GetTotXS(double &tot, double &el, double &in) const;
189:
190:   static std::complex<double> SingleAmpElastic(double s, double t, int type);
191:   std::complex<double>        S3Density(double bt) const;
192:   std::complex<double>        S3Screening(double kt2) const;
193:
194:   // Get random number of cut Pomerons
195:   template <typename T>
196:   void S3GetRandomCutsBt(unsigned int &m, double &bt, T &rng) {
197:     const double STEP = (Numerics.MaxBT - Numerics.MinBT) / Numerics.NumberBT;
198:
199:     // Numerical integral loop over impact parameter (b_t) space
200:     // C++11, thread_local is also static
201:     thread_local std::uniform_real_distribution<double>      flat(0, 1);
202:     thread_local std::uniform_int_distribution<unsigned int> randbt(0, Numerics.NumberBT);
203:     thread_local std::uniform_int_distribution<unsigned int> randm(1, MCUT - 1);  // 1,2,...
204:
205:     // Acceptance-Rejection
206:     unsigned int n = 0;
207:     while (true) {
208:       // Draw random impact parameter flat
209:       n = randbt(rng);
210:
211:       const int value = randm(rng);
212:       if (flat(rng) < P_array[value][n]) {
213:         m = value;  // m cut Pomerons
214:         break;
215:       }
216:     }
217:     bt = Numerics.MinBT + n * STEP;
218:   }
219:
220:   // Get random number of cut Pomerons
221:   template <typename T>
222:   unsigned int S3GetRandomCuts(T &rng) {
223:     // Random integer from [1,MNins-1]
224:     // C++11, thread_local is also static
225:     thread_local std::uniform_real_distribution<double>      flat(0, 1);
226:     thread_local std::uniform_int_distribution<unsigned int> RANDI(1, P_cut.size() - 1);
227:
228:     // Acceptance-Rejection
229:     while (true) {
230:       const unsigned int m = RANDI(rng);
231:       if (flat(rng) < P_cut[m]) {
232:         return m;  // m cut Pomerons
233:       }
234:     }
235:   }
236:
237:   // Arrays
238:   IArray1D MBT;
239:   IArray1D MSA;
240:
241:   // Parameters
242:   MEikonalNumerics Numerics;
243:
244:  private:
245:   static const unsigned int MCUT = 25;  // Maximum number of cut Pomerons
246:
247:   // S3 soft survival initialized
248:   bool S3INIT = false;
249:
```

./include/Graniitti/MEikonal.h 4/4

```
250:   void S3CalculateArray(IArray1D &add, std::complex<double> (MEikonal::*f)(double) const);
251:   void S3CalcXS();
252:   void S3InitCutPomerons();
253:
254:   // Mandelstam s
255:   double s = 0.0;
256:
257:   // Initial state
258:   std::vector<gra::MParticle> INITIALSTATE;
259:
260:   // Integrated eikonal based total cross sections
261:   double sigma_tot  = 0.0;
262:   double sigma_el   = 0.0;
263:   double sigma_inel = 0.0;
264:
265:   // Cut Pomeron probabilities
266:   std::vector<double>              P_cut;
267:   std::vector<std::vector<double>> P_array;
268:
269:   // Two-Channel eikonal based cross sections
270:   double sigma_diff[4] = {0.0};
271:
272:   static const unsigned int X = 0;
273:   static const unsigned int Y = 1;
274: };
275:
276: }  // namespace gra
277:
278: #endif
```

./include/Graniitti/MTensor.h 1/2

```
  1: // Minimal tensor class
  2: //
  3: // Example: (rank-6 tensor with dim-4 per dimension)
  4: //
  5: // MTensor<double> tensor = MTensor({4,4,4,4,4,4}, 0.0);
  6: // tensor({0,3,2,0,1,2}) = 1.0;
  7: //
  8: //
  9: // (c) 2017-2020 Mikael Mieskolainen
 10: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
 11:
 12: #ifndef MTENSOR_H
 13: #define MTENSOR_H
 14:
 15: #include <initializer_list>
 16: #include <iomanip>
 17: #include <iostream>
 18:
 19: namespace gra {
 20: template <typename T>
 21: class MTensor {
 22:  public:
 23:   MTensor() { data = nullptr; }
 24:   MTensor(const std::vector<std::size_t> &newdim) {
 25:     dim  = newdim;
 26:     data = new T[Prod(dim)];
 27:     std::fill(data, data + Prod(dim), T());  // No initialization
 28:   }
 29:   MTensor(const std::vector<std::size_t> &newdim, T value) {
 30:     dim  = newdim;
 31:     data = new T[Prod(dim)];
 32:     std::fill(data, data + Prod(dim), T(value));  // Initialization
 33:   }
 34:   ~MTensor() {
 35:     // delete dynamically allocated memory
 36:     delete[] data;
 37:   }
 38:   // Copy constructor
 39:   MTensor(const MTensor &a) {
 40:     dim  = a.dim;
 41:     data = new T[Prod(dim)];
 42:
 43:     // Copy all elements
 44:     Copy(a);
 45:   }
 46:   // Assignment operator
 47:   MTensor &operator=(const MTensor &rhs) {
 48:     if (data != rhs.data && rhs.data != nullptr) {
 49:       ReSize(rhs.dim);
 50:       Copy(rhs);
 51:     }
 52:     return *this;
 53:   }
 54:   T &operator()(const std::vector<size_t> &ind) {
 55:     const std::size_t i = Index(ind);
 56:     return data[i];
 57:   }
 58:   T &operator()(const std::vector<size_t> &ind) const {
 59:     const std::size_t i = Index(ind);
 60:     return data[i];
 61:   }
 62:   // Size operators
 63:   std::size_t size(std::size_t ind) const { return dim[ind]; }
 64:
 65:   /*
 66:     // Print full tensor
 67:     void Print() {
 68:
 69:       X(:,:,1,1,1) =
 70:       std::vector<std::size_t> state(dim.size()-2, 0);
 71:
 72:       while (true) {
 73:         for (std::size_t k = 2; k < dim.size(); ++k) {
 74:         }
 75:       }
 76:
 77:       for (std::size_t i = 0; i < dim[0]; ++i) {
 78:         for (std::size_t j = 0; j < dim[1]; ++j) {
 79:
 80:           const std::size_t ind = Index({i,j,})
 81:           std::cout <<
 82:         }
 83:       }
```

./include/Graniitti/MTensor.h 2/2

```
 84:       std::cout << data[i]
 85:     }
 86:   */
 87:
 88:  private:
 89:   // Multidimensional indexing algorithm
 90:   // Row-major order
 91:   std::size_t Index(const std::vector<size_t> &ind) const {
 92:     if (ind.size() != dim.size()) {
 93:       throw std::invalid_argument(
 94:             "MTensor:: Error: Index vector with rank = " + std::to_string(ind.size()) +
 95:             " c.f. Tensor has rank " + std::to_string(dim.size()));
 96:     }
 97:     std::size_t sum = 0;
 98:     for (std::size_t i = 0; i < ind.size(); ++i) {
 99:       if (ind[i] >= dim[i]) {
100:         throw std::invalid_argument("MTensor:: Error: Input index " + std::to_string(i) +
101:                                     " over bounds: " + std::to_string(ind[i]) +
102:                                     " >= " + std::to_string(dim[i]));
103:       }
104:       std::size_t product = 1;
105:       for (std::size_t j = i + 1; j < ind.size(); ++j) { product *= dim[j]; }
106:       sum += product * ind[i];
107:     }
108:     return sum;
109:   }
110:   // Copy data from a to *this (after ReSize)
111:   void Copy(const MTensor &a) {
112:     T *p = data + Prod(dim);
113:     T *q = a.data + Prod(dim);
114:     while (p > data) { *--p = *--q; }
115:   }
116:   // Re-Allocate
117:   void ReSize(std::vector<std::size_t> newdim) {
118:     if (data != nullptr) { delete[] data; }
119:     dim  = newdim;
120:     data = new T[Prod(newdim)];
121:   }
122:   // Product to get array volume (memory)
123:   std::size_t Prod(const std::vector<std::size_t> &x) const {
124:     std::size_t product = 1;
125:     for (std::size_t i = 0; i < x.size(); ++i) { product *= x[i]; }
126:     return product;
127:   }
128:
129:   // Dimensions
130:   std::vector<std::size_t> dim;
131:
132:   T *data;
133: };
134:
135: }  // namespace gra
136:
137: #endif
```

./include/Graniitti/MParticle.h 1/2

```
 1: // Particle and branch objects [HEADER ONLY file]
 2: //
 3: // (c) 2017-2020 Mikael Mieskolainen
 4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
 5:
 6: #ifndef MPARTICLE_H
 7: #define MPARTICLE_H
 8:
 9: // C++
10: #include <complex>
11: #include <random>
12: #include <valarray>
13: #include <vector>
14:
15: #include "Graniitti/MAux.h"
16: #include "Graniitti/MCW.h"
17: #include "Graniitti/MHELMatrix.h"
18: #include "Graniitti/MMatrix.h"
19:
20: namespace gra {
21:
22:
23: // Particle class
24: struct MParticle {
25:   // These values are read from the PDG-file
26:   std::string name;
27:   int         pdg      = 0;  // PDG code
28:   int         chargeX3 = 0;  // Q x 3
29:   int         spinX2   = 0;  // J x 2
30:   int         color    = 0;  // QCD color code
31:
32:   double mass  = 0.0;
33:   double width = 0.0;
34:   double tau   = 0.0;  // hbar / width
35:
36:   // J^PC
37:   int         P    = 1;      // Default even P-parity
38:   int         C    = 0;      // Default zero C-parity (e.g. fermions do not have)
39:   unsigned int L    = 0;      // For Mesons/Baryons
40:   bool        glue = false;  // Glueball state
41:
42:   void setPCL(int _P, int _C, unsigned int _L) {
43:     P = _P;
44:     C = _C;
45:     L = _L;
46:   }
47:
48:   // Width cut (in Breit-Wigner sampling)
49:   double wcut = 0.0;
50:
51:   void print() const {
52:     std::cout << " NAME:   " << name << std::endl;
53:     std::cout << " ID:     " << pdg << std::endl;
54:     std::cout << " M:      " << mass << std::endl;
55:     std::cout << " W:      " << width << std::endl;
56:     std::cout << " J^PC:   " << aux::Spin2XtoString(spinX2) << "^" << aux::ParityToString(P)
57:               << aux::ParityToString(C) << std::endl
58:               << std::endl;
59:   }
60: };
61:
62:
63: // Recursive decay tree branch
64: struct MDecayBranch {
65:   MDecayBranch() {
66:     f = MMatrix<std::complex<double>>(1, 1, 1.0);  // Unit element
67:   }
68:
69:   // Offshell mass picked event by event
70:   double m_offshell = 0.0;
71:
72:   MParticle                  p;              // PDG particle
73:   M4Vec                      p4;             // 4-momentum
74:   std::vector<MDecayBranch> legs;            // Daughters
75:   M4Vec                      decay_position; // Decay 4-position
76:   gra::HELMatrix             hel;            // Decay helicity information
77:
78:   // Used with helicity amplitudes
79:   MMatrix<std::complex<double>> f;
80:
81:   // MC weight container
82:   gra::kinematics::MCW W;
83:
```

./include/Graniitti/MParticle.h 2/2

```
84:   // Active in the factorized phase space product (by default, no)
85:   // This is controlled by the spesific amplitudes
86:   bool PS_active = false;
87:
88:   // Decay tree current level
89:   int depth = 0;
90: };
91:
92: }  // namespace gra
93:
94: #endif
```

./include/Graniitti/MContinuum.h 1/1

```
 1: // Continuum 2->N phase space class
 2: //
 3: // (c) 2017-2020 Mikael Mieskolainen
 4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
 5:
 6: #ifndef MCONTINUUM_H
 7: #define MCONTINUUM_H
 8:
 9: // C++
10: #include <complex>
11: #include <vector>
12:
13: // Own
14: #include "Graniitti/M4Vec.h"
15: #include "Graniitti/MAux.h"
16: #include "Graniitti/MMatrix.h"
17: #include "Graniitti/MProcess.h"
18: #include "Graniitti/MSpin.h"
19:
20: // HepMC33
21: #include "HepMC3/GenEvent.h"
22:
23: namespace gra {
24: class MContinuum : public MProcess {
25:  public:
26:   MContinuum();
27:   MContinuum(std::string process, const std::vector<aux::OneCMD> &syntax);
28:   virtual ~MContinuum();
29:
30:   void post_Constructor();
31:
32:   double operator()(const std::vector<double> &randvec, AuxIntData &aux) {
33:     return EventWeight(randvec, aux);
34:   }
35:   double EventWeight(const std::vector<double> &randvec, AuxIntData &aux);
36:   bool   EventRecord(HepMC3::GenEvent &evt);
37:   void   PrintInit(bool silent) const;
38:
39:  private:
40:   void Initialize();
41:   bool LoopKinematics(const std::vector<double> &p1p, const std::vector<double> &p2p);
42:   bool FiducialCuts() const;
43:
44:   // 3*N-4 dimensional phase space, 2->N
45:   bool BNRandomKin(unsigned int Nf, const std::vector<double> &randvec);
46:   bool BNBuildKin(unsigned int Nf, double pt1, double pt2, double phi1, double phi2,
47:                   const std::vector<double> &kt, const std::vector<double> &phi,
48:                   const std::vector<double> &y, double m1, double m2);
49:
50:   void BLinearSystem(std::vector<M4Vec> &p, const std::vector<M4Vec> &q, const M4Vec &p1f,
51:                      const M4Vec &p2f) const;
52:
53:   double BNIntegralVolume() const;
54:   double BNPhaseSpaceWeight() const;
55:
56:   // Auxialary (kt) vectors
57:   std::vector<M4Vec> pkt_;
58: };
59:
60: }  // namespace gra
61:
62: #endif
```

./include/Graniitti/MResonance.h 1/2

```
 1: // Resonance container class
 2: //
 3: // (c) 2017-2020 Mikael Mieskolainen
 4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
 5:
 6: #ifndef MRESONANCE_H
 7: #define MRESONANCE_H
 8:
 9: // C++
10: #include <complex>
11: #include <map>
12: #include <vector>
13:
14: // Own
15: #include "Graniitti/M4Vec.h"
16: #include "Graniitti/MHELMatrix.h"
17: #include "Graniitti/MParticle.h"
18:
19: namespace gra {
20:
21: // Resonance parameters
22: class PARAM_RES {
23:  public:
24:   PARAM_RES() {}
25:   void PrintParam(double sqrts) const {
26:     std::cout << rang::style::bold << "Custom resonance parameters:" << rang::style::reset
27:               << std::endl
28:               << std::endl;
29:
30:     printf("- PDG ID:      %d \n", p.pdg);
31:     printf("- Mass M0:     %0.5f GeV \n", p.mass);
32:     printf("- Width W:     %0.5f GeV \n", p.width);
33:     printf("- Spin Jx2:    %d \n", p.spinX2);
34:     printf("- Parity P:    %d \n", p.P);
35:     std::cout << std::endl;
36:
37:     printf("<Production> \n");
38:     printf("- Effective vertex constant g_i:  %0.1E x exp(i x %0.1f) \n", std::abs(g), std::arg(g));
39:     printf("- Form factor parameter: %0.2f \n", g_FF);
40:
41:     std::cout << std::endl;
42:     printf("<Decay> \n");
43:     printf("- BR:           %0.3E\n", hel.BR);
44:     printf("- Effective vertex constant g_f:  %0.3E", hel.g_decay);
45:     std::cout << std::endl << std::endl;
46:
47:     if (p.spinX2 != 0) {
48:       std::cout << "- Polarization Lorentz frame: " << FRAME << std::endl;
49:       std::cout << std::endl;
50:       std::cout << "- Spin polarization density matrix [rho]:" << std::endl << std::endl;
51:
52:       // Print elements
53:       for (std::size_t i = 0; i < rho.size_row(); ++i) {
54:         for (std::size_t j = 0; j < rho.size_col(); ++j) {
55:           std::string delim = (j < rho.size_col() - 1) ? ", " : "";
56:           printf("%6.3f+i%6.3f%s ", std::real(rho[i][j]), std::imag(rho[i][j]), delim.c_str());
57:         }
58:         std::cout << std::endl;
59:       }
60:       std::cout << std::endl << std::endl;
61:       // printf("Density matrix von Neumann entropy S = %0.3f
62:       // \n\n", MSpin::VonNeumannEntropy(rho));
63:     }
64:   }
65:
66:   // Particle class
67:   MParticle p;
68:
69:   // ------------------------------------------------------------------
70:   // Tensor Pomeron couplings
71:   std::vector<double> g_Tensor;  // Production couplings
72:   // ------------------------------------------------------------------
73:
74:   // (Complex) production coupling constant
75:   std::complex<double> g = 0.0;
76:
77:   // Form factor parameter
78:   double g_FF = 0.0;
79:
80:   // Breit-Wigner type
81:   int BW = 0;
82:
83:   // 2->1 (generation spin correlation), 1->2 (decay spin correlations)
```

./include/Graniitti/MResonance.h 2/2

```
  bool SPINGEN = true;
  bool SPINDEC = true;

  // Lorentz frame of the spin distribution
  std::string FRAME = "null";

  // Maximum sliding pomeron helicity
  int JMAX = 0;

  // Spin-density matrix (constant)
  MMatrix<std::complex<double>> rho;

  // ------------------------------------------------------------------------
  // Helicity and decay amplitude information
  HELMatrix hel;
  // ------------------------------------------------------------------------
};

}  // namespace gra

#endif
```

./include/Graniitti/MGlobals.h 1/1

```
// GRANIITTI Monte Carlo all global variables collected here
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

#ifndef MGLOBALS_H
#define MGLOBALS_H

// Own
#include "Graniitti/MSudakov.h"

// LHAPDF
#include "LHAPDF/LHAPDF.h"

namespace gra {
// ============================================================================
// These variables are initialized by MGraniitti.cc

// Model tune
extern std::string MODELPARAM;

// Multithreading lock
extern std::mutex g_mutex;

// For multithreaded VEGAS, to handle the exceptions from forked threads
extern std::exception_ptr globalExceptionPtr;

// ============================================================================

}  // namespace gra

#endif
```

./include/Graniitti/MTimer.h 1/1

```
// Timer class
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

#ifndef MTIMER_H
#define MTIMER_H

// C++
#include <chrono>

namespace gra {
class MTimer {
 public:
  // Declare explicit, means here no conversion to bool allowed
  explicit MTimer(bool reset = true) {
    if (reset) { Reset(); }
  }
  // << operator
  template <typename T, typename Traits>
  friend std::basic_ostream<T, Traits> &operator<<(std::basic_ostream<T, Traits> &out,
                                                   const MTimer &                timer) {
    return out << timer.Elapsed().count();
  }
  // Time in sec
  double ElapsedSec() const { return Elapsed().count() / 1000.0; }
  // Time in msec
  std::chrono::milliseconds Elapsed() const {
    return std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::high_resolution_clock::now() - start);
  }
  // Reset timer
  void Reset() { start = std::chrono::high_resolution_clock::now(); }

 private:
  std::chrono::high_resolution_clock::time_point start;
};

}  // namespace gra

#endif
```

./include/Graniitti/MForm.h 1/2

```
// Form factors, structure functions, Regge trajectories etc. parametrizations
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

#ifndef MFORM_H
#define MFORM_H

// C++
#include <complex>
#include <map>
#include <vector>

// Own
#include "Graniitti/M4Vec.h"
#include "Graniitti/MAux.h"
#include "Graniitti/MKinematics.h"
#include "Graniitti/MParticle.h"
#include "Graniitti/MRandom.h"
#include "Graniitti/MResonance.h"

// Libraries
#include "json.hpp"

namespace gra {

// Model parameters
namespace PARAM_SOFT {
// Pomeron trajectory
extern double DELTA_P;
extern double ALPHA_P;

// Couplings
extern double gN_P;
extern double gN_O;

extern double g3P;
extern double gamma;

// Proton form factor
extern double fc1;
extern double fc2;
extern double fc3;

extern bool ODDERON_ON;

void        PrintParam();
std::string GetHashString();

void        ReadParameters(const std::string& modelfile);
extern bool initialized;
}  // namespace PARAM_SOFT

namespace PARAM_STRUCTURE {
extern std::string F2;
extern std::string EM;
extern std::string QED_alpha;

void        ReadParameters(const std::string& modelfile);
extern bool initialized;
}  // namespace PARAM_STRUCTURE

// Flat (DEBUG) amplitude parameters
namespace PARAM_FLAT {
extern double b;

void        ReadParameters(const std::string& modelfile);
extern bool initialized;
}  // namespace PARAM_FLAT

// Forward proton excitation
namespace PARAM_NSTAR {
extern std::string          fragment;
extern std::vector<double> rc;

void        ReadParameters(const std::string& modelfile);
extern bool initialized;
}  // namespace PARAM_NSTAR

namespace form {

// Read resonance
gra::PARAM_RES ReadResonance(const std::string& resparam_str, MRandom& rng);
```

```
 84:
 85: // Regge signature
 86: std::complex<double> ReggeEta(double alpha_t, double sigma);
 87: std::complex<double> ReggeEtaLinear(double t, double alpha_t0, double ap, double sigma);
 88:
 89: // Proton form factor, elastic and inelastic
 90: double S3F(double t);
 91: double S3FINEL(double t, double M2);
 92:
 93: // Proton structure functions
 94: double F2xQ2(double xi, double Q2);
 95: double F1xQ2(double xi, double Q2);
 96:
 97: // Pomeron trajectory
 98: double S3PomAlpha(double t);
 99:
100: // Pion loop insert
101: double S3HPL(double tau, double t);
102:
103: // t-integrated collinear EPA flux
104: double D2Flux(double x);
105:
106: // Coherent gamma flux
107: double F1(double Q2);
108: double F2(double Q2);
109: double G_M(double Q2);
110: double G_E(double Q2);
111:
112: double G_M_DIPOLE(double Q2);
113: double G_E_DIPOLE(double Q2);
114:
115: double G_M_KELLY(double Q2);
116: double G_E_KELLY(double Q2);
117:
118: double CohFlux(double x, double t, double pt);
119: double IncohFlux(double x, double t, double pt, double M2);
120:
121: double mu_ratio();
122:
123: // Breit-Wigner functions
124: double deltaBWxsec(double shat, double M0, double Gamma);
125: double deltaBWamp(double shat, double M0, double Gamma);
126:
127: std::complex<double> CBW(const gra::LORENTZSCALAR& lts, const gra::PARAM_RES& resonance);
128: std::complex<double> CBW_FW(double m2, double M0, double Gamma);
129: std::complex<double> CBW_RW(double m2, double M0, double Gamma);
130: std::complex<double> CBW_BF(double m2, double M0, double Gamma, int J, double mA, double mB);
131: std::complex<double> CBW_JR(double m2, double M0, double Gamma, double J);
132:
133: }  // namespace form
134: }  // namespace gra
135:
136: #endif
```



```
  1: // 2D histogram class
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: #ifndef MH2_H
  7: #define MH2_H
  8:
  9: // C++
 10: #include <complex>
 11: #include <vector>
 12:
 13: // Own
 14: #include "Graniitti/MMatrix.h"
 15:
 16: namespace gra {
 17: class MH2 {
 18:  public:
 19:   MH2(int xbins, double xmin, double xmax, int ybins, double ymin, double ymax,
 20:       std::string namestr = "noname");
 21:
 22:   MH2(int xbins, int ybins, std::string namestr = "noname");
 23:   MH2();
 24:   ~MH2();
 25:
 26:   void ResetBounds(int xbins, double xmin, double xmax, int ybins, double ymin, double ymax);
 27:
 28:   // Get full histogram data
 29:   MMatrix<double>        GetWeights() const { return weights; }
 30:   MMatrix<double>        GetWeights2() const { return weights2; }
 31:   MMatrix<long long int> GetCounts() const { return counts; }
 32:
 33:   void                   Fill(double xvalue, double yvalue);
 34:   void                   Fill(double xvalue, double yvalue, double weight);
 35:   void                   Clear();
 36:   std::pair<double, double> WeightMeanAndError() const;
 37:
 38:   double GetMeanX(int power) const;
 39:   double GetMeanY(int power) const;
 40:
 41:   double        SumWeights() const;
 42:   double        SumWeights2() const;
 43:   long long int SumBinCounts() const;
 44:   long long int FillCount() const { return fills; }
 45:
 46:   double        GetBinWeight(int xbin, int ybin) const;
 47:   long long int GetBinCount(int xbin, int ybin) const;
 48:   void          GetBinIdx(double xvalue, double yvalue, int &xbin, int &ybin) const;
 49:   double        GetMaxWeight() const;
 50:   double        GetMinWeight() const;
 51:   void          Print() const;
 52:   double        ShannonEntropy() const;
 53:
 54:   // Set logarithmic binning
 55:   // (user needs to take care that XMIN and XMAX > 0)
 56:   void SetLogX() { LOGX = true; }
 57:   void SetLogY() { LOGX = true; }
 58:   void SetLogXY() {
 59:     LOGX = true;
 60:     LOGY = true;
 61:   }
 62:
 63:   // Overload + operator to add two histograms
 64:   MH2 operator+(const MH2 &rhs) {
 65:     if ((this->XBINS != rhs.XBINS) || (this->YBINS != rhs.YBINS)) {
 66:       throw std::domain_error("MH2 + operator: Histograms with different number of bins");
 67:     }
 68:     MH2 h(this->XBINS, this->XMIN, this->XMAX, this->YBINS, this->YMIN, this->YMAX, this->name);
 69:
 70:     h.fills     = this->fills + rhs.fills;
 71:     h.underflow = (this->underflow[0] + rhs.underflow[0], this->underflow[1] + rhs.underflow[1]);
 72:     h.overflow  = (this->overflow[0] + rhs.overflow[0], this->overflow[1] + rhs.overflow[1]);
 73:     h.nanflow   = this->nanflow + rhs.nanflow;
 74:
 75:     // DATA
 76:     h.weights  = this->weights + rhs.weights;
 77:     h.weights2 = this->weights2 + rhs.weights2;
 78:     h.counts   = this->counts + rhs.counts;
 79:
 80:     return h;
 81:   }
 82:
 83:   // AUTOBUFFSIZE for autorange buffer size
```



```
 84:   void SetAutoBuffSize(int n) { AUTOBUFFSIZE = n; }
 85:   void FlushBuffer();
 86:
 87:   // Symmetric bounds
 88:   void SetAutoSymmetry(const std::vector<bool> &in) {
 89:     if (in.size() != 2) {
 90:       throw std::invalid_argument("MH2::SetAutoSymmetry: Input should be size 2 boolean vector");
 91:     }
 92:     AUTOSYMMETRY = in;
 93:   }
 94:
 95:   void GetBounds(int &xbins, double &xmin, double &xmax, int &ybins, double &ymin,
 96:                  double &ymax) const {
 97:     xbins = XBINS;
 98:     xmin  = XMIN;
 99:     xmax  = XMAX;
100:     ybins = YBINS;
101:     ymin  = YMIN;
102:     ymax  = YMAX;
103:   }
104:
105:   void FuseBuffer(const MH2 &rhs) {
106:     buff_values.insert(buff_values.end(), rhs.buff_values.begin(), rhs.buff_values.end());
107:     buff_weights.insert(buff_weights.end(), rhs.buff_weights.begin(), rhs.buff_weights.end());
108:   }
109:
110:   // Keep it public for buffer fusion
111:   std::vector<std::vector<double>> buff_values;
112:   std::vector<double>              buff_weights;
113:
114:  private:
115:   std::string name;  // Histogram name
116:
117:   // ------------------------------------------------------------
118:   // For autorange
119:
120:   bool              FILLBUFF     = false;
121:   int               AUTOBUFFSIZE = 100000;  // Default AUTOBUFFSIZE
122:   std::vector<bool> AUTOSYMMETRY = {false, false};
123:   // ------------------------------------------------------------
124:
125:   // Boundary conditions
126:   double XMIN  = 0.0;
127:   double XMAX  = 0.0;
128:   int    XBINS = 0;
129:
130:   double YMIN  = 0.0;
131:   double YMAX  = 0.0;
132:   int    YBINS = 0;
133:
134:   // Number of underflow and overflow counts
135:   long long int              fills    = 0;
136:   std::vector<long long int> overflow  = {0, 0};
137:   std::vector<long long int> underflow = {0, 0};
138:   long long int              nanflow   = 0;
139:
140:   // Logarithmic binning
141:   bool LOGX = false;
142:   bool LOGY = false;
143:   bool ValidBin(int xbin, int ybin) const;
144:   int  GetIdx(double value, double minval, double maxval, int nbins, bool logbins) const;
145:
146:   // Weights (in unweighted case weights = counts)
147:   MMatrix<double> weights;
148:   MMatrix<double> weights2;
149:
150:   // Counts
151:   MMatrix<long long int> counts;
152: };
153:
154: }  // namespace gra
155:
156: #endif
```



```
  1: // Gamma amplitudes
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: #ifndef MGAMMA_H
  7: #define MGAMMA_H
  8:
  9: // C++
 10: #include <complex>
 11: #include <random>
 12: #include <vector>
 13:
 14: // Own
 15: #include "Graniitti/M4Vec.h"
 16: #include "Graniitti/MAmplitudes.h"
 17: #include "Graniitti/MKinematics.h"
 18:
 19: namespace gra {
 20:
 21: // Monopole wave functions and parameters
 22: namespace PARAM_MONOPOLE {
 23:
 24: extern bool printed;      // Printing called
 25: extern bool initialized;  // For lazy initialization
 26:
 27: extern int         En;        // Bound state energy level
 28: extern double      M0;        // Monopole mass
 29: extern double      Gamma0;    // Monopolium width
 30: extern std::string coupling;  // Coupling scenarios
 31: extern int         gn;        // Monopole charge n = 1,2,3,...
 32:
 33: double EnergyMP(double n);
 34: double GammaMP(double n, double alpha_g);
 35: double PsiMP(double n);
 36:
 37: void PrintParameters(double sqrts);
 38: void ReadParameters(const std::string &modelfile);
 39:
 40: }  // namespace PARAM_MONOPOLE
 41:
 42:
 43: class MGamma : public MAmplitudes {
 44:  public:
 45:   MGamma(gra::LORENTZSCALAR &lts, const std::string &modelfile);
 46:   ~MGamma() {}
 47:
 48:   // yy->resonance X
 49:   double yyX(gra::LORENTZSCALAR &lts, gra::PARAM_RES &resonance) const;
 50:
 51:   // yy->lepton pair, or monopole antimonopole amplitude
 52:   double yyffbar(gra::LORENTZSCALAR &lts);
 53:
 54:   // yy->SM Higgs
 55:   double yyHiggs(gra::LORENTZSCALAR &lts) const;
 56:
 57:   // yy->monopolium
 58:   double yyMP(gra::LORENTZSCALAR &lts) const;
 59:
 60:  protected:
 61: };
 62:
 63: }  // namespace gra
 64:
 65: #endif
```

./include/Graniitti/MSudakov.h 1/3

```
// Shuvaev PDF and Sudakov suppression class
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

#ifndef MSUDAKOV_H
#define MSUDAKOV_H

// C++
#include <complex>
#include <memory>
#include <vector>

// LHAPDF
#include "LHAPDF/LHAPDF.h"

// Own
#include "Graniitti/MAux.h"
#include "Graniitti/MMath.h"
#include "Graniitti/MTimer.h"

// Libraries
#include "json.hpp"

namespace gra {

struct MSudakovNumerics {
  double       q2_MIN = 0.0;
  double       q2_MAX = 0.0;
  double       M_MIN  = 0.0;
  double       M_MAX  = 0.0;
  double       x_MIN  = 0.0;
  const double x_MAX  = 1.0 - 1E-9;

  // Numerical integral discretization [SET HERE]
  const unsigned int SudakovIntegralN = 1000;
  const unsigned int ShuvaevIntegralN = 400;

  // Logarithmic stepping true/false
  std::vector<bool> SUDA_log_ON;
  std::vector<bool> SHUV_log_ON;

  // Number of discrete intervals
  std::vector<unsigned int> SUDA_N;
  std::vector<unsigned int> SHUV_N;

  // CMS energy
  double sqrts = 0.0;

  bool DEBUG = false;

  void ReadParameters() {
    // Read and parse
    using json                = nlohmann::json;
    const std::string inputfile = gra::aux::GetBasePath(2) + "/modeldata/" + "NUMERICS.json";
    const std::string data      = gra::aux::GetInputData(inputfile);
    json              j;

    try {
      j = json::parse(data);

      // JSON block identifier
      const std::string XID = "NUMERICS_SUDAKOV";

      q2_MIN = j.at(XID).at("q2_MIN");
      q2_MAX = j.at(XID).at("q2_MAX");
      M_MIN  = j.at(XID).at("M_MIN");
      // M_MAX = j.at(XID).at("M_MAX"); // Post-Setup
      // x_MIN = j.at(XID).at("x_MIN"); // Post-Setup
      // x_MAX = j.at(XID).at("x_MAX"); // Set above

      // Logarithmic stepping true/false [assign needed for <cast>]
      std::vector<bool> xx = j.at(XID).at("SUDA").at("log_ON");
      SUDA_log_ON          = xx;
      std::vector<bool> yy = j.at(XID).at("SHUV").at("log_ON");
      SHUV_log_ON          = yy;

      // Number of node points [assign needed for <cast>]
      std::vector<unsigned int> nn = j.at(XID).at("SUDA").at("N");
      SUDA_N                       = nn;
      std::vector<unsigned int> mm = j.at(XID).at("SHUV").at("N");
      SHUV_N                       = mm;
```

./include/Graniitti/MSudakov.h 2/3

```
      // Discretization
      // ShuvaevIntegralN = j.at(XID).at("ShuvaevIntegralN");
      // SudakovIntegralN = j.at(XID).at("SudakovIntegralN");

      DEBUG = j.at(XID).at("DEBUG");
    } catch (...) {
      std::string str = "MSudakovNumerics::ReadParameters: Error parsing " + inputfile +
                        " (Check for extra/missing commas)";
      throw std::invalid_argument(str);
    }
  }
};


// Interpolation container
class IArray2D {
 public:
  IArray2D(){};
  ~IArray2D(){};

  std::string  name[2];
  double       MIN[2]   = {0};
  double       MAX[2]   = {0};
  unsigned int N[2]     = {0};
  double       STEP[2]  = {0};
  bool         islog[2] = {false};

  // Setup discretization
  void Set(unsigned int VAR, std::string _name, double _min, double _max, double _N,
           double _logarithmic) {
    // Out of index
    if (VAR > 1) {
      throw std::invalid_argument("IArray2D::Set: Error: VAR = " + std::to_string(VAR) + " > 1");
    }

    // AT least two intervals
    if (_N < 2) {
      throw std::invalid_argument("IArray2D::Set: Error: N = " + std::to_string(_N) + " < 2");
    }

    // Sanity
    if (_min >= _max) {
      throw std::invalid_argument("IArray2D::Set: Error: Variable " + _name + " MIN = " +
                                  std::to_string(_min) + " >= MAX = " + std::to_string(_max));
    }

    // Sanity
    if (_logarithmic && _min < 1e-9) {
      throw std::invalid_argument(
          "IArray2D::Set: Error: Variable " + _name +
          " is using logarithmic stepping with boundary MIN = " + std::to_string(_min));
    }
    islog[VAR] = _logarithmic;

    // Logarithm taken here!
    MIN[VAR] = islog[VAR] ? std::log(_min) : _min;
    MAX[VAR] = islog[VAR] ? std::log(_max) : _max;
    N[VAR]   = _N;

    STEP[VAR] = (MAX[VAR] - MIN[VAR]) / N[VAR];
    name[VAR] = _name;
  }

  // Call this last
  void InitArray() {
    // Note N+1 !
    F = std::vector<std::vector<std::vector<double>>>(
        N[0] + 1, std::vector<std::vector<double>>(N[1] + 1, std::vector<double>(4, 0.0)));
  }

  // N+1 per dimension!
  std::vector<std::vector<std::vector<double>>> F;

  // CMS energy (needed for boundary conditions)
  double sqrts = 0.0;

  std::string GetHashString() const {
    std::string str = std::to_string(islog[0]) + std::to_string(islog[1]) + std::to_string(MIN[0]) +
                      std::to_string(MIN[1]) + std::to_string(MAX[0]) + std::to_string(MAX[1]) +
                      std::to_string(N[0]) + std::to_string(N[1]) + std::to_string(sqrts);
    return str;
  }
```

./include/Graniitti/MSudakov.h 3/3

```
  bool                       WriteArray(const std::string &filename, bool overwrite) const;
  bool                       ReadArray(const std::string &filename);
  std::pair<double, double> Interpolate2D(double A, double B) const;
};

// Sudakov suppression and skewed pdf
//
// Take care when copying this class - there is a pointer to LHAPDF
class MSudakov {
 public:
  MSudakov();
  ~MSudakov();

  void Init(double sqrts_in, const std::string &PDFSET, bool init_arrays = true);
  void InitLHAPDF(const std::string &PDFSET);
  std::pair<double, double> Shuvaev_H(double q2, double x);
  std::pair<double, double> Sudakov_T(double qt2, double M);

  double fg_xQ2M(double x, double q2, double M) const;
  double AlphaS_Q2(double q2) const;
  double NumFlavor(double q2) const;
  double xg_xQ2(double x, double Q2) const;
  void   TestPDF() const;

  bool initialized = false;

 private:
  int init_trials = 0;

  void   InitArrays();
  double diff_xg_xQ2_wrt_Q2(double x, double q2) const;
  double AP_gg(double delta) const;
  double AP_qg(double delta, double qt2) const;
  void   CalculateArray(IArray2D &arr, std::pair<double, double> (MSudakov::*f)(double, double));

  std::string  PDFSETNAME;
  LHAPDF::PDF *PdfPtr = nullptr;

  IArray2D veto;  // Sudakov veto
  IArray2D spdf;  // Shuvaev pdf

  MSudakovNumerics Numerics;  // Numerics
};

}  // namespace gra

#endif
```

./include/Graniitti/MParton.h 1/1

```
// Simple parton model 2->2 phase space class
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

#ifndef MPARTON_H
#define MPARTON_H

// C++
#include <complex>
#include <random>
#include <vector>

// Own
#include "Graniitti/M4Vec.h"
#include "Graniitti/MAux.h"
#include "Graniitti/MMatrix.h"
#include "Graniitti/MProcess.h"
#include "Graniitti/MSpin.h"

// HepMC33
#include "HepMC3/GenEvent.h"

namespace gra {
class MParton : public MProcess {
 public:
  MParton();
  MParton(std::string process, const std::vector<aux::OneCMD> &syntax);
  virtual ~MParton();

  void post_Constructor();

  double operator()(const std::vector<double> &randvec, AuxIntData &aux) {
    return EventWeight(randvec, aux);
  }
  double EventWeight(const std::vector<double> &randvec, AuxIntData &aux);
  bool   EventRecord(HepMC3::GenEvent &evt);
  void   PrintInit(bool silent) const;

 private:
  void Initialize();
  bool LoopKinematics(const std::vector<double> &p1p, const std::vector<double> &p2p);
  bool FiducialCuts() const;

  // 2->2 dim phase space
  bool B2RandomKin(const std::vector<double> &randvec);
  bool B2BuildKin(double xbj1, double xbj2);
  void B2RecordEvent(HepMC3::GenEvent &evt);

  double B2IntegralVolume() const;
  double B2PhaseSpaceWeight() const;

  void DecayWidthPS(double &exact) const;
};

}  // namespace gra

#endif
```

./include/Graniitti/MRegge.h 1/2

```
 1: // Regge Amplitudes
 2: //
 3: // (c) 2017-2020 Mikael Mieskolainen
 4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
 5:
 6: #ifndef MREGGE_H
 7: #define MREGGE_H
 8:
 9: // C++
10: #include <complex>
11: #include <random>
12: #include <vector>
13:
14: // Own
15: #include "Graniitti/M4Vec.h"
16: #include "Graniitti/MKinematics.h"
17:
18:
19: namespace gra {
20:
21: // Regge amplitude parameters
22: namespace PARAM_REGGE {
23:
24: extern bool initialized;
25:
26: extern std::vector<double> a0;
27: extern std::vector<double> ap;
28: extern std::vector<double> sgn;
29:
30: extern double s0;
31:
32: extern int    offshellFF;
33: extern double b_EXP;
34: extern double a_OREAR;
35: extern double b_OREAR;
36: extern double b_POW;
37:
38: extern bool reggeize;
39:
40: extern double omega;
41:
42: // Meson/Baryon Couplings and Pomeron, Reggeon, Reggeon exchanges
43: extern std::vector<double> c;  // coupling
44: extern std::vector<bool>   n;  // on/off
45:
46: void PrintParam();
47: void ReadParameters(int PDG, const std::string &modelfile);
48:
49: std::complex<double> JPC_CS_coupling(const gra::LORENTZSCALAR &lts,
50:                                      const gra::PARAM_RES &    resonance);
51:
52: // Amplitude form factors
53: double Proton_FF(double tprime, double b);
54: double Meson_FF(double that, double M2);
55: double Baryon_FF(double that, double M2);
56: double ResonanceFormFactor(double shat, double M2, double S0);
57:
58: // Propagators
59: double               Meson_prop(double that, double M2);
60: double               Baryon_prop(double that, double M2);
61: std::complex<double> FSI_prop(double t_hat, double M2);
62: }  // namespace PARAM_REGGE
63:
64: // Matrix element dimension: " GeV^" << -(2*external_legs - 8)
65: class MRegge {
66:  public:
67:   MRegge(gra::LORENTZSCALAR &lts, const std::string &modelfile);
68:   ~MRegge() {}
69:
70:   // Regge amplitudes
71:   std::complex<double> ME8(gra::LORENTZSCALAR &lts) const;
72:   std::complex<double> ME6(gra::LORENTZSCALAR &lts) const;
73:   std::complex<double> ME4(gra::LORENTZSCALAR &lts, double sign) const;
74:   std::complex<double> ME2(gra::LORENTZSCALAR &lts, int mode) const;
75:   std::complex<double> ME3(gra::LORENTZSCALAR &lts, gra::PARAM_RES &resonance) const;
76:   std::complex<double> ME3ODD(gra::LORENTZSCALAR &lts, gra::PARAM_RES &resonance) const;
77:
78:   std::complex<double> ME3HEL(gra::LORENTZSCALAR &lts, gra::PARAM_RES &resonance) const;
79:   std::complex<double> PhotoME3(gra::LORENTZSCALAR &lts, gra::PARAM_RES &resonance) const;
80:
81:   void PomPomProtonVertex(const gra::LORENTZSCALAR &lts, double &FF_A, double &FF_B) const;
82:
83:   // Constructor amplitude leg permutations
```

./include/Graniitti/MRegge.h 2/2

```
 84:   void ConstructPerm(int type);
 85:
 86:   // Regge propagators
 87:   std::complex<double> PropOnly(double s, double t) const;
 88:   std::complex<double> OdderonProp(double s, double t) const;
 89:   std::complex<double> PhotoProp(double s, double t, double m2, bool excite,
 90:                                  double M2_forward) const;
 91:
 92:   // Helicity functions
 93:   double               g_Vertex(double t, double lambda_i, double lambda_f) const;
 94:   std::complex<double> gjk_Vertex(double t1, double t2, double dphi, int lambda_h, int J, int P,
 95:                                   int JMAX) const;
 96:   double               gammaLambda(double t1, double t2, double m1, double m2) const;
 97:   int                  xi3(int J, int P, int P_i, int sigma_i, int P_k, int sigma_k) const;
 98:
 99:  private:
100:   // Amplitude permutations
101:   std::vector<std::vector<int>> permutations4;
102:   std::vector<std::vector<int>> permutations6;
103: };
104:
105: }  // namespace gra
106:
107: #endif
```

./include/Graniitti/M4Vec.h 1/4

```
 1: // 4-vectors [HEADER ONLY class]
 2: // with standard metric (+,-,-,-) and MC initialization convention (px,py,pz,e)
 3: //
 4: // However, note that ^,* operators index in "normal textbook convention",
 5: // to be more streamlined with Lorent index contractions of amplitudes
 6: //
 7: //  p^0 =  E
 8: //  p^1 =  px
 9: //  p^2 =  py
10: //  p^3 =  pz
11: //
12: //  p%0 =  E
13: //  p%1 = -px
14: //  p%2 = -py
15: //  p%3 = -pz
16: //
17: // (c) 2017-2020 Mikael Mieskolainen
18: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
19:
20: #ifndef M4VEC_H
21: #define M4VEC_H
22:
23: #include <complex>
24: #include <iostream>
25: #include <vector>
26:
27: namespace gra {
28: // ------------------------------------------------------------------------
29: // C++17 allows class template argument default deduction without brackets
30: //
31: // M4Vec a;
32: // M4Vec<> b;
33: // M4Vec<double> c;
34: //
35: // a,b,c are all valid.
36: //
37: // template <typename T = double>
38: //
39: // Complex 4-vectors could be implemented this way, TBD.
40: // ------------------------------------------------------------------------
41:
42: class M4Vec {
43:  public:
44:   // All default to zero
45:   M4Vec() { k = {0.0, 0.0, 0.0, 0.0}; }
46:   // Initialize
47:   M4Vec(double x, double y, double z, double t) { k = {t, x, y, z}; }
48:   // Copy constructor
49:   M4Vec(const M4Vec &rhs) { k = rhs.k; }
50:
51:   // SET methods
52:   void SetPx(double v) { k[X_] = v; }
53:   void SetPy(double v) { k[Y_] = v; }
54:   void SetPz(double v) { k[Z_] = v; }
55:   void SetE(double v) { k[E_] = v; }
56:
57:   void SetX(double v) { k[X_] = v; }
58:   void SetY(double v) { k[Y_] = v; }
59:   void SetZ(double v) { k[Z_] = v; }
60:   void SetT(double v) { k[E_] = v; }
61:
62:   void Set(double x, double y, double z, double t) {
63:     k[X_] = x;
64:     k[Y_] = y;
65:     k[Z_] = z;
66:     k[E_] = t;
67:   }
68:
69:   void SetPxPyPzM(double x, double y, double z, double m) {
70:     k[X_] = x;
71:     k[Y_] = y;
72:     k[Z_] = z;
73:     k[E_] = msqrt(P3mod2() + m * m);
74:   }
75:
76:   void SetPxPyPz(double x, double y, double z) {
77:     k[X_] = x;
78:     k[Y_] = y;
79:     k[Z_] = z;
80:   }
81:
82:   void SetPxPy(double x, double y) {
83:     k[X_] = x;
```

./include/Graniitti/M4Vec.h 2/4

```
 84:     k[Y_] = y;
 85:   }
 86:
 87:   void SetPzE(double z, double e) {
 88:     k[Z_] = z;
 89:     k[E_] = e;
 90:   }
 91:
 92:   void SetP3(const std::vector<double> &vec) {
 93:     if (vec.size() != 3) { throw std::invalid_argument("M4Vec::SetP3: input should be 3-vector"); }
 94:     k[X_] = vec[0];
 95:     k[Y_] = vec[1];
 96:     k[Z_] = vec[2];
 97:   }
 98:
 99:   // Apply metric tensor: eta_{\mu\nu} k^\nu = k_\mu
100:   void Flip3() {
101:     k[X_] = -k[X_];
102:     k[Y_] = -k[Y_];
103:     k[Z_] = -k[Z_];
104:   }
105:
106:   // GET methods
107:   double Px() const { return k[X_]; }
108:   double Py() const { return k[Y_]; }
109:   double Pz() const { return k[Z_]; }
110:   double E() const { return k[E_]; }
111:
112:   double X() const { return k[X_]; }
113:   double Y() const { return k[Y_]; }
114:   double Z() const { return k[Z_]; }
115:   double T() const { return k[E_]; }
116:
117:   // Return 3-vector
118:   std::vector<double> P3() const { return {k[X_], k[Y_], k[Z_]}; }
119:
120:   // ALGEBRA methods
121:
122:   // Space-time invariants
123:   double Invariant() const { return E() * E() - P3mod2(); }
124:   double M2() const { return Invariant(); }
125:   double M() const { return (M2() > 0.0) ? msqrt(M2()) : -msqrt(-M2()); }
126:
127:   // gamma = E/m = 1/sqrt(1-v^2/c^2) = 1/sqrt(1-beta^2)
128:   double Gamma() const { return E() / M(); }
129:   double Beta() const { return P3mod() / E(); }
130:
131:   // Transverse 2-vector norm and norm^2
132:   double Perp() const { return Pt(); }
133:   double Perp2() const { return Pt2(); }
134:   double Pt() const { return msqrt(Pt2()); }
135:   double Pt2() const { return k[X_] * k[X_] + k[Y_] * k[Y_]; }
136:
137:   // Total 3-vector norm and norm^2
138:   double P3mod() const { return msqrt(P3mod2()); }
139:   double P3mod2() const { return k[X_] * k[X_] + k[Y_] * k[Y_] + k[Z_] * k[Z_]; }
140:
141:   // Transverse mass (invariant under boost in z-direction)
142:   double Mt() const { return msqrt(Mt2()); }
143:   double Mt2() const { return M2() + Pt2(); }
144:
145:   // Coincides with transverse mass for single particle
146:   double Et() const { return Mt(); }
147:   double Et2() const { return Mt2(); }
148:
149:   // Angles
150:   double Phi() const {
151:     return (Px() == 0.0 && Py() == 0.0) ? 0.0 : std::atan2(Py(), Px());
152:   }  // y / x, range [-PI,PI]
153:   double Theta() const {
154:     return (Px() == 0.0 && Py() == 0.0 && Pz() == 0.0) ? 0.0 : std::atan2(Pt(), Pz());
155:   }  // |Pt| / z
156:   double CosTheta() const { return std::cos(Theta()); }
157:
158:   // Pseudorapidity and rapidity (boost) in z-direction
159:   double Eta() const { return 0.5 * std::log((P3mod() + Pz()) / (P3mod() - Pz())); }
160:   double Rap() const { return 0.5 * std::log((E() + Pz()) / (E() - Pz())); }
161:
162:   // Lightcone variable: k_+ = E + p_z
163:   double LightconePos() const { return E() + Pz(); }
164:
165:   // Lightcone variable: k_- = E - p_z
166:   double LightconeNeg() const { return E() - Pz(); }
```

./include/Graniitti/M4Vec.h 3/4

```
167:
168:   // SPINOR-HELICITY CO-VARIABLES
169:
170:   std::complex<double> ComplexPt() const { return Px() + std::complex<double>(0, 1) * Py(); }
171:   std::complex<double> ExpCPhi() const {
172:     return ComplexPt() / msqrt(LightconePos() * LightconeNeg());
173:   }
174:
175:   // 2-BODY ALGEBRA
176:
177:   // Minkowski 4-product
178:   double DotM(const M4Vec &rhs) const {
179:     return E() * rhs.E() - (Px() * rhs.Px() + Py() * rhs.Py() + Pz() * rhs.Pz());
180:   }
181:
182:   // 3-vector dot product
183:   double Dot3(const M4Vec &rhs) const {
184:     return Px() * rhs.Px() + Py() * rhs.Py() + Pz() * rhs.Pz();
185:   }
186:
187:   // Transverse 2-vector dot product
188:   double DotPt(const M4Vec &rhs) const { return Px() * rhs.Px() + Py() * rhs.Py(); }
189:
190:   // 3-vector cross product (return vector with 0 energy/time)
191:   M4Vec Cross3(const M4Vec &rhs) const {
192:     M4Vec a(Py() * rhs.Pz() - Pz() * rhs.Py(), Pz() * rhs.Px() - Px() * rhs.Pz(),
193:             Px() * rhs.Py() - Py() * rhs.Px(), 0.0);
194:     return a;
195:   }
196:
197:   // Azimuth angle difference between [-PI,PI]
198:   double DeltaPhi(const M4Vec &v) const {
199:     double D = Phi() - v.Phi();
200:     while (D >= PI) { D -= 2.0 * PI; }
201:     while (D < -PI) { D += 2.0 * PI; }
202:     return D;
203:   }
204:
205:   // Azimuth angle between [0,PI]
206:   double DeltaPhiAbs(const M4Vec &v) const { return std::abs(DeltaPhi(v)); }
207:
208:   // OPERATORS
209:   double  operator[](size_t mu) const { return k[mu]; }  // for reading only
210:   double &operator[](size_t mu) { return k[mu]; }        // for substituting
211:
212:   // Access operator in normal contravariant (upper index) indexing
213:   double operator^(size_t mu) const { return k[mu]; }
214:
215:   // Access operator with simultaneous lowering with metric (covariant index)
216:   double operator%(size_t mu) const {
217:     if (mu == E_) {
218:       return k[E_];
219:     } else {
220:       return -k[mu];
221:     }
222:     throw std::out_of_range("M4Vec::operator %% Index out of bounds!");
223:   }
224:
225:   // Minkowski scalar product
226:   double operator*(const M4Vec &rhs) const { return DotM(rhs); }
227:
228:   // 4-vector + 4-vector
229:   M4Vec operator+(const M4Vec &rhs) const {
230:     return M4Vec(k[X_] + rhs.k[X_], k[Y_] + rhs.k[Y_], k[Z_] + rhs.k[Z_], k[E_] + rhs.k[E_]);
231:   }
232:   M4Vec operator-(const M4Vec &rhs) const {
233:     return M4Vec(k[X_] - rhs.k[X_], k[Y_] - rhs.k[Y_], k[Z_] - rhs.k[Z_], k[E_] - rhs.k[E_]);
234:   }
235:
236:   // Flip sign of all components
237:   M4Vec operator-() const { return M4Vec(-k[X_], -k[Y_], -k[Z_], -k[E_]); }
238:
239:   // 4-vector */ scalar
240:   M4Vec operator*(const double rhs) const {
241:     return M4Vec(k[X_] * rhs, k[Y_] * rhs, k[Z_] * rhs, k[E_] * rhs);
242:   }
243:   M4Vec operator/(const double rhs) const {
244:     return M4Vec(k[X_] / rhs, k[Y_] / rhs, k[Z_] / rhs, k[E_] / rhs);
245:   }
246:
247:   // Comparison
248:   bool operator==(const M4Vec &rhs) const {
249:     const double EPS = 1e-10;
```

./include/Graniitti/M4Vec.h 4/4

```
250:     return std::abs(k[E_] - rhs.k[E_]) < EPS && std::abs(k[X_] - rhs.k[X_]) < EPS &&
251:            std::abs(k[Y_] - rhs.k[Y_]) < EPS && std::abs(k[Z_] - rhs.k[Z_]) < EPS;
252:   }
253:   bool operator!=(const M4Vec &rhs) const { return !(*this == rhs); }
254:
255:   // 4-vector += 4-vector
256:   void operator+=(const M4Vec &rhs) {
257:     k[E_] += rhs.k[E_];
258:     k[X_] += rhs.k[X_];
259:     k[Y_] += rhs.k[Y_];
260:     k[Z_] += rhs.k[Z_];
261:   }
262:   void operator-=(const M4Vec &rhs) {
263:     k[E_] -= rhs.k[E_];
264:     k[X_] -= rhs.k[X_];
265:     k[Y_] -= rhs.k[Y_];
266:     k[Z_] -= rhs.k[Z_];
267:   }
268:
269:   // 4-vector *= scalar
270:   void operator*=(const double rhs) {
271:     k[E_] *= rhs;
272:     k[X_] *= rhs;
273:     k[Y_] *= rhs;
274:     k[Z_] *= rhs;
275:   }
276:   void operator/=(const double rhs) {
277:     k[E_] /= rhs;
278:     k[X_] /= rhs;
279:     k[Y_] /= rhs;
280:     k[Z_] /= rhs;
281:   }
282:
283:   void Print(const std::string name = "") const {
284:     std::cout << "M4Vec::" << name << " Px (X): " << Px() << ", Py (Y): " << Py()
285:               << ", Pz (Z): " << Pz() << ", E (T): " << E() << ", M (S): " << M()
286:               << ", theta: " << Theta() << ", phi: " << Phi() << std::endl;
287:   }
288:
289:   // Particle 4-position starting starting propagation from (0,0,0,0)
290:   // - p is the particle 4-momentum in the lab frame
291:   // - tau0 is the particle flight time in its rest frame
292:   M4Vec PropagatePosition(double tau0, double scale) const {
293:     // Flight time in the lab frame
294:     const double gamma = Gamma();
295:     const double tau   = gamma * tau0;
296:
297:     // Velocity
298:     const double beta = Beta();
299:
300:     // End point 4-position
301:     return M4Vec(tau * c * beta * Px() / P3mod() * scale, tau * c * beta * Py() / P3mod() * scale,
302:                  tau * c * beta * Pz() / P3mod() * scale, tau * c * scale);
303:   }
304:
305:  private:
306:   static constexpr const double PI =
307:       3.14159265358979323846264338327950288419716939937510582097494 4L;
308:
309:   // speed of light, c = [m/s] (EXACT/DEFINITION)
310:   static constexpr const double c = 2.99792458E8;
311:
312:   // Indices
313:   static const int E_ = 0;
314:   static const int X_ = 1;
315:   static const int Y_ = 2;
316:   static const int Z_ = 3;
317:
318:   // Safe sqrt
319:   double msqrt(double x) const { return std::sqrt(std::max(0.0, x)); }
320:   // 4-vector
321:   std::vector<double> k;
322: };
323:
324: }  // namespace gra
325:
326: #endif
```

./include/Graniitti/MAmplitudes.h 1/1

```
1: // All external amplitudes (matrix elements) from MadGraph etc. collected here
2: //
3: // (c) 2017-2020 Mikael Mieskolainen
4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
5:
6: #ifndef MAMPLITUDES_H
7: #define MAMPLITUDES_H
8:
9: // C++
10: #include <complex>
11: #include <random>
12: #include <vector>
13:
14: // Own
15: #include "Graniitti/M4Vec.h"
16: #include "Graniitti/MKinematics.h"
17:
18: // MadGraph
19: #include "Graniitti/Amplitude/AMP_MG5_gg_gg.h"
20: #include "Graniitti/Amplitude/AMP_MG5_qq_qqg.h"
21: #include "Graniitti/Amplitude/AMP_MG5_yy_ll.h"
22: #include "Graniitti/Amplitude/AMP_MG5_yy_ww.h"
23: #include "Graniitti/Amplitude/AMP_MG5_yy_ww_evev.h"
24:
25:
26: namespace gra {
27:
28: // Matrix element dimension: " GeV^" << -(2*external_legs - 8)
29: class MAmplitudes {
30:  public:
31:   MAmplitudes() {}
32:   ~MAmplitudes() {}
33:
34:   AMP_MG5_yy_ww       AmpMG5_yy_ww;
35:   AMP_MG5_yy_ll       AmpMG5_yy_ll;
36:   AMP_MG5_yy_ww_evev  AmpMG5_yy_ww_evev;
37:   AMP_MG5_gg_gg       AmpMG5_gg_gg;
38:   AMP_MG5_gg_ggg      AmpMG5_gg_ggg;
39:
40:  protected:
41: };
42:
43: }  // namespace gra
44:
45: #endif
```

./include/Graniitti/MHELMatrix.h 1/1

```
1: // Kinematic objects and helicity coupling structures [HEADER ONLY file]
2: //
3: // (c) 2017-2020 Mikael Mieskolainen
4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
5:
6: #ifndef MHELMATRIX_H
7: #define MHELMATRIX_H
8:
9: // C++
10: #include <complex>
11: #include <vector>
12:
13: #include "Graniitti/MMatrix.h"
14:
15: namespace gra {
16:
17:
18: // Decay helicity amplitude information
19: struct HELMatrix {
20:   void InitAlphaToZero() {
21:     const unsigned int N = 20;
22:     alpha               = MMatrix<std::complex<double>>(N, N, 0.0);
23:     alpha_set           = MMatrix<bool>(N, N, true);
24:   }
25:
26:   // Decay amplitude matrix (calculated by SU(2) decomposition routines)
27:   gra::MMatrix<std::complex<double>> T;
28:
29:   // Helicity amplitude decay ls-couplings (constant)
30:   gra::MMatrix<std::complex<double>> alpha;
31:   gra::MMatrix<bool>                 alpha_set;
32:
33:   // Parity conservation
34:   bool P_conservation = true;
35:
36:   // ------------------------------------------------------------------------
37:   // Tensor Pomeron couplings
38:   std::vector<double> g_decay_tensor;  // Decay couplings
39:   // ------------------------------------------------------------------------
40:
41:   // Branching Ratio to a particular decay
42:   double BR      = 1.0;
43:   double g_decay = 1.0;  // Equivalent decay coupling
44: };
45:
46:
47: }  // namespace gra
48:
49: #endif
```

```
// Templated real valued or complex 1D-histogram class
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

#ifndef MH1_H
#define MH1_H

// C++
#include <complex>
#include <vector>

namespace gra {
template <class T>
class MH1 {
 public:
  MH1(int xbins, double xmin, double xmax, std::string namestr = "noname");
  MH1(int xbins, std::string namestr = "noname");
  MH1();
  ~MH1();

  void Fill(double xvalue);
  void Fill(double xvalue, T weight);
  void Clear();

  void RawOutput() const;

  long long int FillCount() const { return fills; }
  long long int SumBinCounts() const;
  double        GetMean(int power) const;

  std::pair<double, double> WeightMeanAndError() const;

  // Probability density
  std::vector<double> GetProbDensity() const {
    double sum = 0;
    for (std::size_t i = 0; i < weights.size(); ++i) { sum += GetPositiveDefinite(i); }
    std::vector<double> p(weights.size(), 0.0);
    if (sum > 0) {
      for (std::size_t i = 0; i < weights.size(); ++i) { p[i] = GetPositiveDefinite(i) / sum; }
    }
    return p;
  }

  // Get full histogram data
  std::vector<T>              GetWeights() const { return weights; }
  std::vector<T>              GetWeights2() const { return weights2; }
  std::vector<long long int> GetCounts() const { return counts; }

  void GetXPositiveDefinite(std::valarray<double> &x, std::valarray<double> &y) const;

  T SumWeights() const;
  T SumWeights2() const;

  T      GetBinWeight(int idx) const;
  T      GetBinWeight2(int idx) const;
  double GetBinError(int idx) const;

  double GetMaxWeight() const;
  double GetMinWeight() const;

  long long int GetBinCount(int idx) const;
  double        GetPositiveDefinite(int i) const;

  double GetBinXVal(int idx, int boundary = 0) const;
  void   GetBinIdx(double xvalue, int &idx);
  void   Print(double width = 1.25) const;  // default argument

  void ResetBounds(int xbins);
  void ResetBounds(int xbins, double xmin, double xmax);

  // Set logarithmic binning
  void SetLogX() {
    if (XMIN < 1e-9) {
      throw std::invalid_argument(
          "MH1::SetLogX: Error: Minimum boundary XMIN = " + std::to_string(XMIN) + " < 0");
    }
    LOGX = true;
  }

  // Number of bins
  std::size_t XBins() const { return XBINS; }
```



```
  // Copy constructor (DEFAULT is fine)
  // MH1<T> MH1(const MH1<T>& obj) {}

  // Overload + operator to add two histograms
  MH1<T> operator+(const MH1<T> &rhs) {
    if (this->XBINS != rhs.XBINS) {
      throw std::domain_error("MH1<T> + operator: Histograms with different number of bins");
    }

    MH1<T> h(this->XBINS, this->XMIN, this->XMAX, this->name);

    h.fills     = this->fills + rhs.fills;
    h.underflow = this->underflow + rhs.underflow;
    h.overflow  = this->overflow + rhs.overflow;
    h.nanflow   = this->nanflow + rhs.nanflow;

    // DATA
    h.weights  = this->weights;
    h.weights2 = this->weights2;
    h.counts   = this->counts;
    for (std::size_t i = 0; i < h.weights.size(); ++i) {
      h.weights[i] += rhs.weights[i];
      h.weights2[i] += rhs.weights2[i];
      h.counts[i] += rhs.counts[i];
    }
    return h;
  }

  // Overload - operator to subtract two histograms
  MH1<T> operator-(const MH1<T> &rhs) {
    if (this->XBINS != rhs.XBINS) {
      throw std::domain_error("MH1<T> - operator: Histograms with different number of bins");
    }

    MH1<T> h(this->XBINS, this->XMIN, this->XMAX, this->name);

    h.fills     = this->fills - rhs.fills;
    h.underflow = this->underflow - rhs.underflow;
    h.overflow  = this->overflow - rhs.overflow;
    h.nanflow   = this->nanflow - rhs.nanflow;

    // DATA
    h.weights  = this->weights;
    h.weights2 = this->weights2;
    h.counts   = this->counts;
    for (std::size_t i = 0; i < h.weights.size(); ++i) {
      h.weights[i] -= rhs.weights[i];
      h.weights2[i] -= rhs.weights2[i];
      h.counts[i] -= rhs.counts[i];
    }
    return h;
  }

  // Overload * operator to multiply two histograms
  MH1<T> operator*(const MH1<T> &rhs) {
    if (this->XBINS != rhs.XBINS) {
      throw std::domain_error("MH1<T> * operator: Histograms with different number of bins");
    }

    MH1<T> h(this->XBINS, this->XMIN, this->XMAX, this->name);

    h.fills     = this->fills;
    h.underflow = this->underflow;
    h.overflow  = this->overflow;
    h.nanflow   = this->nanflow;

    // DATA
    h.weights  = this->weights;
    h.weights2 = this->weights2;
    h.counts   = this->counts;
    for (std::size_t i = 0; i < h.weights.size(); ++i) {
      h.weights[i] *= rhs.weights[i];
      h.weights2[i] *= rhs.weights2[i];
      h.counts[i] *= rhs.counts[i];
    }
    return h;
  }

  // Overload / operator to divide two histograms
  MH1<T> operator/(const MH1<T> &rhs) {
    if (this->XBINS != rhs.XBINS) {
      throw std::domain_error("MH1<T> / operator: Histograms with different number of bins");
    }
```



```
    MH1<T> h(this->XBINS, this->XMIN, this->XMAX, this->name);

    h.fills     = this->fills;
    h.underflow = this->underflow;
    h.overflow  = this->overflow;
    h.nanflow   = this->nanflow;

    // DATA
    h.weights  = this->weights;
    h.weights2 = this->weights2;
    h.counts   = this->counts;
    for (std::size_t i = 0; i < h.weights.size(); ++i) {
      h.weights[i]  = (std::abs(rhs.weights[i]) > 0) ? h.weights[i] / rhs.weights[i] : 0;
      h.weights2[i] = (std::abs(rhs.weights2[i]) > 0) ? h.weights2[i] / rhs.weights2[i] : 0;
      h.counts[i]   = (rhs.counts[i] > 0) ? h.counts[i] / rhs.counts[i] : 0;
    }
    return h;
  }

  // AUTOBUFFSIZE for autorange buffer size
  void SetAutoBuffSize(int n) { AUTOBUFFSIZE = n; }
  void FlushBuffer();

  // Symmetric bounds
  void SetAutoSymmetry(bool in) { AUTOSYMMETRY = in; }

  // Get histogram bounds
  void GetBounds(int &xbins, double &xmin, double &xmax) const {
    xbins = XBINS;
    xmin  = XMIN;
    xmax  = XMAX;
  }

  void FuseBuffer(const MH1<T> &rhs) {
    buff_values.insert(buff_values.end(), rhs.buff_values.begin(), rhs.buff_values.end());
    buff_weights.insert(buff_weights.end(), rhs.buff_weights.begin(), rhs.buff_weights.end());
  }

  // Keep it public for buffer fusion
  std::vector<double> buff_values;
  std::vector<T>      buff_weights;

 private:
  std::string name;  // Histogram name

  // ------------------------------------------------------------
  // For autorange
  // ------------------------------------------------------------
  bool FILLBUFF     = false;
  int  AUTOBUFFSIZE = 100000;  // Default AUTOBUFFSIZE
  bool AUTOSYMMETRY = false;
  // ------------------------------------------------------------

  // Boundary conditions
  double XMIN  = 0.0;
  double XMAX  = 0.0;
  int    XBINS = 0;

  // Number of fills, overflow and underflow counts
  long long int fills     = 0;
  long long int overflow  = 0;
  long long int underflow = 0;
  long long int nanflow   = 0;

  // weights (in unweighted case weights = counts)
  std::vector<T>              weights;
  std::vector<T>              weights2;
  std::vector<long long int> counts;  // counts

  // Logarithmic binning
  bool LOGX = false;

  bool ValidBin(int idx) const;
  int  GetIdx(double value, double minval, double maxval, int nbins, bool logbins) const;

  // Comparing do we have double or std::complex<double>
  constexpr bool IsReal() const { return std::is_same<T, double>::value; }
  constexpr bool IsComplex() const { return std::is_same<T, std::complex<double>>::value; }
};

// Include the implementation
//#include "MH1.tcc"
```



```
}  // namespace gra

#endif
```

```
 1: // Dirac gamma algebra routines
 2: //
 3: // (c) 2017-2020 Mikael Mieskolainen
 4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
 5:
 6: #ifndef MDIRAC_H
 7: #define MDIRAC_H
 8:
 9: // C++
10: #include <complex>
11: #include <random>
12: #include <vector>
13:
14: // Own
15: #include "Graniitti/M4Vec.h"
16: #include "Graniitti/MKinematics.h"
17: #include "Graniitti/MMath.h"
18: #include "Graniitti/MMatrix.h"
19:
20: // FTensor
21: #include "FTensor.hpp"
22:
23: namespace gra {
24: class MDirac {
25:  public:
26:   MDirac();
27:   MDirac(const std::string &basis);
28:   ~MDirac() {}
29:
30:   void InitGammaMatrices(const std::string &basis);
31:
32:   // Polarization vectors/tensors
33:   FTensor::Tensor1<std::complex<double>, 4>     EpsSpin1(const M4Vec &k, int m) const;
34:   FTensor::Tensor1<std::complex<double>, 4>     EpsMassiveSpin1(const M4Vec &k, int m) const;
35:   FTensor::Tensor2<std::complex<double>, 4, 4> EpsMassiveSpin2(const M4Vec &k, int m) const;
36:
37:   // Spin/spinor state collectors
38:   std::array<std::vector<std::complex<double>>, 2>          SpinorStates(const M4Vec &      p,
39:                                                                          const std::string &type) const;
40:   std::array<FTensor::Tensor1<std::complex<double>, 4>, 2> MasslessSpin1States(
41:       const M4Vec &p, const std::string &type, bool INDEX_UP = true) const;
42:   std::array<FTensor::Tensor1<std::complex<double>, 4>, 3> MassiveSpin1States(
43:       const M4Vec &p, const std::string &type, bool INDEX_UP = true) const;
44:
45:   // Helicity spinors
46:   std::vector<std::complex<double>> XiSpinor(const M4Vec &p, int helicity) const;
47:   std::vector<std::complex<double>> uHelChiral(const M4Vec &p, int helicity) const;
48:   std::vector<std::complex<double>> vHelChiral(const M4Vec &p, int helicity) const;
49:
50:   std::vector<std::complex<double>> uHelDirac(const M4Vec &p, int helicity) const;
51:   std::vector<std::complex<double>> vHelDirac(const M4Vec &p, int helicity) const;
52:
53:   // Dirac adjoint
54:   std::vector<std::complex<double>> Bar(const std::vector<std::complex<double>> &spinor) const;
55:
56:   // Feynman slash matrix operator
57:   MMatrix<std::complex<double>> FSlash(const M4Vec &a) const;
58:
59:   // Propagators
60:   FTensor::Tensor2<std::complex<double>, 4, 4> iD_y(const double q2) const;
61:   MMatrix<std::complex<double>>                iD_F(const M4Vec &q, double m) const;
62:
63:   // Dirac spinors
64:   std::vector<std::complex<double>> uDirac(const M4Vec &p, int spin) const;
65:   std::vector<std::complex<double>> vDirac(const M4Vec &p, int spin) const;
66:
67:   // Spinor-Helicity style methods
68:   std::complex<double>              sProd(const M4Vec &p1, const M4Vec &p2, int helicity) const;
69:   std::vector<std::complex<double>> uGauge(const M4Vec &p, int helicity) const;
70:   std::vector<std::complex<double>> vGauge(const M4Vec &p, int helicity) const;
71:
72:   // Charge conjugate operator
73:   MMatrix<std::complex<double>> C_up() const;
74:
75:   // Right and left chiral projectors
76:   MMatrix<std::complex<double>> PR() const;
77:   MMatrix<std::complex<double>> PL() const;
78:
79:   // Angular Momentum operators
80:   MMatrix<std::complex<double>> J_operator(unsigned int i) const;
81:
82:   // Test functions
83:   double TestGammaAntiCommutation() const;
```



```
 84:   double TestSpinorHELimit(const M4Vec &p1, const M4Vec &p2) const;
 85:   double TestSpinorComplete(const M4Vec &p, const std::string &type, const std::string &mode) const;
 86:   double TestMassiveSpin1Complete(const M4Vec &k) const;
 87:   double TestFSlashFSlash(const M4Vec &p) const;
 88:
 89:   // Kronecker delta
 90:   double Delta(int i, int j) const { return (i == j) ? 1.0 : 0.0; }
 91:
 92:   // Transform matrix from the Weyl to Dirac basis by an operator sandwich:
 93:   //
 94:   // \gamma^\mu_Dirac = S \gamma_{Weyl}^\mu S^{-1}
 95:   //
 96:   // S = S^{-1} <=> So this matrix works also from Dirac to Weyl basis
 97:   //
 98:   // Spinor Transformation:
 99:   //
100:   // u_Dirac = S u_Weyl
101:   // v_Dirac = S v_Weyl
102:   //
103:   // S = 1/\sqrt{2}(1 + y^5y^0)
104:   //
105:   MMatrix<std::complex<double>> S_basis = {
106:       std::vector<std::complex<double>>{1.0 / std::sqrt(2.0), 0.0, 1.0 / std::sqrt(2.0), 0.0},
107:       std::vector<std::complex<double>>{0.0, 1.0 / std::sqrt(2.0), 0.0, 1.0 / std::sqrt(2.0)},
108:       std::vector<std::complex<double>>{1.0 / std::sqrt(2.0), 0.0, -1.0 / std::sqrt(2.0), 0.0},
109:       std::vector<std::complex<double>>{0.0, 1.0 / std::sqrt(2.0), 0.0, -1.0 / std::sqrt(2.0)}};
110:
111:   // Pauli matrices
112:   MMatrix<std::complex<double>> sigma_x = {std::vector<std::complex<double>>{0.0, 1.0},
113:                                            std::vector<std::complex<double>>{1.0, 0.0}};
114:
115:   MMatrix<std::complex<double>> sigma_y = {std::vector<std::complex<double>>{0.0, -gra::math::zi},
116:                                            std::vector<std::complex<double>>{gra::math::zi, 0.0}};
117:
118:   MMatrix<std::complex<double>> sigma_z = {std::vector<std::complex<double>>{1.0, 0.0},
119:                                            std::vector<std::complex<double>>{0.0, 1.0}};
120:
121:   // Contravariant (up) and covariant (lo) gamma matrix set
122:   std::vector<MMatrix<std::complex<double>>> gamma_up;
123:   std::vector<MMatrix<std::complex<double>>> gamma_lo;
124:
125:   // Contravariant (up) and covariant (lo) \sigma_{\mu\nu} matrices
126:   std::vector<std::vector<MMatrix<std::complex<double>>>> sigma_up;
127:   std::vector<std::vector<MMatrix<std::complex<double>>>> sigma_lo;
128:
129:   // Indices etc.
130:   std::vector<std::size_t> LI          = {0, 1, 2, 3};
131:   std::vector<int>         SPINORSTATE = {-1, 1};
132:
133:   // Minkowski metric tensor (+,-,-,-) in convention (t,px,py,pz)
134:   MMatrix<double> g = MMatrix<double>(4, 4, "minkowski");
135:
136:   // Identity matrix
137:   MMatrix<std::complex<double>> I4 = MMatrix<std::complex<double>>(4, 4, "eye");
138:
139:  protected:
140:   std::string BASIS = "";  // D for Dirac, C for Chiral
141: };
142:
143: }  // namespace gra
144:
145: #endif
```



```
 1: // (Sub)-Processes and Amplitudes
 2: //
 3: // (c) 2017-2020 Mikael Mieskolainen
 4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
 5:
 6: #ifndef MSUBPROC_H
 7: #define MSUBPROC_H
 8:
 9: // C++
10: #include <complex>
11: #include <random>
12: #include <vector>
13:
14: // Own
15: #include "Graniitti/M4Vec.h"
16: #include "Graniitti/MAux.h"
17: #include "Graniitti/MDurham.h"
18: #include "Graniitti/MFlux.h"
19: #include "Graniitti/MForm.h"
20: #include "Graniitti/MGamma.h"
21: #include "Graniitti/MPDG.h"
22: #include "Graniitti/MRegge.h"
23: #include "Graniitti/MTensorPomeron.h"
24:
25:
26: namespace gra {
27:
28: using math::iabs2;
29: using math::msqrt;
30:
31:
32: // Abstract process base class
33: //
34: // BASIC DESIGN:
35: // Use "late (lazy) construction", that is, objects are pointers and initialized only
36: // just before the event generation.
37: //
38: //
39: class MProc {
40:  public:
41:   MProc(const std::string& i, const std::string& j, const std::vector<std::string>& k) {
42:     ISTATE      = i;
43:     CHANNEL     = j;
44:     DESCRIPTION = k;
45:   };
46:   std::string              ISTATE;
47:   std::string              CHANNEL;
48:   std::vector<std::string> DESCRIPTION;
49:
50:   virtual ~MProc() {}  // Needs to be virtual
51:
52:   virtual double Amp2(gra::LORENTZSCALAR& lts) = 0;
53:
54:   // Processes usable with different fluxes
55:   // 2y ->
56:   double GammaGammaCON(gra::LORENTZSCALAR& lts) {
57:     double amp2 = 0.0;
58:     if (!AssertN(2, lts.decaytree.size())) {
59:       throw std::invalid_argument(ISTATE + "[" + CHANNEL + "] requires 2-body final state");
60:     }
61:
62:     if (AssertN({24, -24), PDGlist(lts))) {
63:       amp2 = Gamma->AmpMG5_yy_ww.CalcAmp2(lts, 0.0);
64:     } else if (AssertLeptonQuarkMonopolePair(PDGlist(lts))) {
65:       amp2 = Gamma->yyffbar(lts);
66:     } else {
67:       ThrowUnknownFinalState();
68:     }
69:
70:     return amp2;
71:   }
72:
73:   // Process class containers
74:   std::unique_ptr<MGamma>         Gamma  = nullptr;
75:   std::unique_ptr<MDurham>        Durham = nullptr;
76:   std::unique_ptr<MRegge>         Regge  = nullptr;
77:   std::unique_ptr<MTensorPomeron> Tensor = nullptr;
78:
79:   // Delayed constructors, so global .json parameters are read from files already at this point
80:   void InitGamma(gra::LORENTZSCALAR& lts) {
81:     if (Gamma == nullptr) { Gamma = make_unique<MGamma>(lts, GetModelFile()); }
82:   }
83:   void InitDurham(gra::LORENTZSCALAR& lts) {
```



```
 84:     if (Durham == nullptr) { Durham = make_unique<MDurham>(lts, GetModelFile()); }
 85:   }
 86:   void InitTensor(gra::LORENTZSCALAR& lts) {
 87:     if (Tensor == nullptr) { Tensor = make_unique<MTensorPomeron>(lts, GetModelFile()); }
 88:   }
 89:   void InitRegge(gra::LORENTZSCALAR& lts) {
 90:     if (Regge == nullptr) { Regge = make_unique<MRegge>(lts, GetModelFile()); }
 91:   }
 92:
 93:   // Return general modelfile
 94:   std::string GetModelFile() const {
 95:     return gra::aux::GetBasePath(2) + "/modeldata/" + gra::MODELPARAM + "/GENERAL.json";
 96:   }
 97:
 98:   // Assert the final state list
 99:   bool AssertN(const std::vector<int>& reference, const std::vector<int>& input) const {
100:     return std::is_permutation(reference.begin(), reference.end(), input.begin());
101:   }
102:
103:   // Assert the number of final states
104:   bool AssertN(int reference, int input) const { return (reference == input) ? true : false; }
105:
106:   // Assert lepton or quark pair
107:   bool AssertLeptonQuarkMonopolePair(const std::vector<int>& input) const {
108:     // PDG identifiers
109:     static const MMatrix<int> X = {{11, -11}, {13, -13}, {15, -15}, {1, -1}, {2, -2},
110:                                    {3, -3},   {4, -4},   {5, -5},   {6, -6}, {992, -992}};
111:
112:     for (std::size_t i = 0; i < X.size_row(); ++i) {
113:       if (AssertN(X[i][0], X[i][1]), input)) { return true; }
114:     }
115:     return false;
116:   }
117:
118:   // Get PDG list
119:   std::vector<int> PDGlist(const LORENTZSCALAR& lts) const {
120:     std::vector<int> L(lts.decaytree.size(), 0);
121:     for (const auto& i : aux::indices(lts.decaytree)) { L[i] = lts.decaytree[i].p.pdg; }
122:     return L;
123:   }
124:
125:   void ThrowUnknownFinalState() const {
126:     throw std::invalid_argument(ISTATE + "[" + CHANNEL + "]" + " with unsupported final state");
127:   }
128: };
129:
130:
131: // ------------------------------------------------------------------------
132: // Gamma-Gamma processes
133: // ------------------------------------------------------------------------
134:
135: class PROC_0 : public MProc {
136:  public:
137:   PROC_0() : MProc("yy", "RES", {"Parametric resonance", "kt-EPA"}) {}
138:   ~PROC_0() {}
139:   virtual double Amp2(gra::LORENTZSCALAR& lts) {
140:     InitGamma(lts);
141:     const double amp2 = Gamma->yyX(lts, lts.RESONANCES.begin()->second);
142:     return flux::ApplyktEPAfluxes(amp2, lts);
143:   }
144: };
145: class PROC_1 : public MProc {
146:  public:
147:   PROC_1() : MProc("yy", "Higgs", {"SM Higgs", "kt-EPA"}) {}
148:   ~PROC_1() {}
149:   virtual double Amp2(gra::LORENTZSCALAR& lts) {
150:     InitGamma(lts);
151:     const double amp2 = Gamma->yyHiggs(lts);
152:     return flux::ApplyktEPAfluxes(amp2, lts);
153:   }
154: };
155: class PROC_2 : public MProc {
156:  public:
157:   PROC_2() : MProc("yy", "monopolium(0)", {"Monopolium (J=0)", "kt-EPA"}) {}
158:   ~PROC_2() {}
159:   virtual double Amp2(gra::LORENTZSCALAR& lts) {
160:     InitGamma(lts);
161:     const double amp2 = Gamma->yyMP(lts);
162:     return flux::ApplyktEPAfluxes(amp2, lts);
163:   }
164: };
165: class PROC_3 : public MProc {
166:  public:
```

```
  PROC_3() : MProc("yy", "CON", {"Continuum 1+1-, qqbar, W+W-, monopolepair", "kt-EPA"}) {}
  ~PROC_3() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitGamma(lts);
    double amp2 = GammaGammaCON(lts);
    return flux::ApplyktEPAfluxes(amp2, lts);
  }
};
class PROC_4 : public MProc {
 public:
  PROC_4() : MProc("yy", "QED", {"Continuum 1+1-, qqbar", "FULL QED"}) {}
  ~PROC_4() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitTensor(lts);

    if (!AssertN(2, lts.decaytree.size())) {
      throw std::invalid_argument(ISTATE + "[" + CHANNEL + "] requires 2-body final state");
    }
    return Tensor->ME4(lts);
  }
};


// ------------------------------------------------------------------------
// Inclusive processes
// ------------------------------------------------------------------------

class PROC_5 : public MProc {
 public:
  PROC_5() : MProc("X", "EL", {"Elastic", "Eikonal Pomeron", "Use with screening loop on"}) {}
  ~PROC_5() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitRegge(lts);
    return abs2(Regge->ME2(lts, 1));
  }
};
class PROC_6 : public MProc {
 public:
  PROC_6() : MProc("X", "SD", {"Single Diffractive", "Triple Pomeron", "With TOY fragmentation"}) {}
  ~PROC_6() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitRegge(lts);
    return abs2(Regge->ME2(lts, 2));
  }
};
class PROC_7 : public MProc {
 public:
  PROC_7() : MProc("X", "DD", {"Double Diffractive", "Triple Pomeron", "With TOY fragmentation"}) {}
  ~PROC_7() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitRegge(lts);
    return abs2(Regge->ME2(lts, 3));
  }
};
class PROC_8 : public MProc {
 public:
  PROC_8()
      : MProc("X", "ND", {"Non-Diffractive", "N-cut soft Pomerons", "With TOY fragmentation"}) {}
  ~PROC_8() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) { return 1.0; }
};


// ------------------------------------------------------------------------
// Pomeron-Pomeron processes
// ------------------------------------------------------------------------

class PROC_9 : public MProc {
 public:
  PROC_9() : MProc("PP", "RES", {"Parametric resonance", "Pomeron"}) {}
  ~PROC_9() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitRegge(lts);
    std::complex<double> A = 0.0;

    // Coherent sum of Resonances (loop over)
    for (auto& x : lts.RESONANCES) {
      const int J = static_cast<int>(x.second.p.spinX2 / 2.0);

      // Gamma-Pomeron for vectors (could be Pomeron-Odderon too)
      if (J == 1 && x.second.p.P == -1) {
        A += Regge->PhotoME3(lts, x.second);
      }
      // Pomeron-Pomeron, J = 0,1,2,... all ok
```



```
      else {
        A += Regge->ME3(lts, x.second);
      }
    }

    // ----------------------------------------------------------------
    // Set for the screening loop
    lts.hamp = {A};
    // ----------------------------------------------------------------

    return abs2(A);
  }
};

class PROC_10 : public MProc {
 public:
  PROC_10()
      : MProc("PP", "RESHEL",
              {"Sliding pomeron helicity amplitudes", "Pomeron", "DEVELOPER ONLY PROCESS!"}) {}
  ~PROC_10() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitRegge(lts);
    std::complex<double> A = 0.0;
    A                      = Regge->ME3HEL(lts, lts.RESONANCES.begin()->second);
    return abs2(A);
  }
};

class PROC_11 : public MProc {
 public:
  PROC_11() : MProc("PP", "RESTENSOR", {"Parametric resonance", "Tensor Pomeron"}) {}
  ~PROC_11() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitTensor(lts);
    return Tensor->ME3(lts);
  }
};

class PROC_12 : public MProc {
 public:
  PROC_12() : MProc("PP", "CONTENSOR", {"Hadron continuum 2-body", "Tensor Pomeron"}) {}
  ~PROC_12() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitTensor(lts);

    if (!AssertN(2, lts.decaytree.size())) {
      throw std::invalid_argument(ISTATE + "[" + CHANNEL + "] requires 2-body final state");
    }
    return Tensor->ME4(lts);
  }
};

class PROC_13 : public MProc {
 public:
  PROC_13()
      : MProc("PP", "CONTENSOR24",
              {"Hadron continuum 2-body > 4-body", "Tensor Pomeron", "DEVELOPER ONLY PROCESS!"}) {}
  ~PROC_13() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitTensor(lts);

    if (!AssertN(2, lts.decaytree.size()) && !AssertN(4, lts.decaytree.size())) {
      throw std::invalid_argument(ISTATE + "[" + CHANNEL +
                                  "] requires 2 > (2x) or 4-body final state");
    }
    return Tensor->ME6(lts);
  }
};

class PROC_14 : public MProc {
 public:
  PROC_14()
      : MProc("PP", "RES+CONTENSOR",
              {"Hadron resonances + continuum 2-body", "Tensor Pomeron / yP"}) {}
  ~PROC_14() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitTensor(lts);

    if (!AssertN(2, lts.decaytree.size())) {
      throw std::invalid_argument(ISTATE + "[" + CHANNEL + "] requires 2-body final state");
    }

    // 1. Evaluate continuum matrix element -> helicity amplitudes to lts.hamp
```



```
    Tensor->ME4(lts);

    // Add to temp vector (init with zero!)
    std::vector<std::complex<double>> tempsum(lts.hamp.size(), 0.0);
    for (const auto& i : aux::indices(lts.hamp)) { tempsum[i] += lts.hamp[i]; }

    // 2. Evaluate resonance matrix elements -> helicity amplitudes to lts.hamp
    if (lts.RESONANCES.size() != 0) {
      // We loop over resonances inside ME3
      Tensor->ME3(lts);
      // Add to temp vector
      for (const auto& i : aux::indices(lts.hamp)) { tempsum[i] += lts.hamp[i]; }
    }
    // ----------------------------------------------------------------
    // Set helicity amplitudes for the screening loop
    lts.hamp = tempsum;
    // ----------------------------------------------------------------

    // Get total amplitude squared 1/4 \sum_h |A_h|^2
    double amp2 = 0.0;
    for (const auto& i : aux::indices(lts.hamp)) { amp2 += gra::math::abs2(lts.hamp[i]); }
    amp2 /= 4;  // Initial state helicity average

    return amp2;
  }
};

class PROC_15 : public MProc {
 public:
  PROC_15() : MProc("PP", "CON", {"Hadron continuum 2/4/6-body", "Pomeron"}) {}
  ~PROC_15() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitRegge(lts);

    std::complex<double> A = 0.0;
    if (AssertN(2, lts.decaytree.size())) {
      A = Regge->ME4(lts, 1);
    } else if (AssertN(4, lts.decaytree.size())) {
      A = Regge->ME6(lts);
    } else if (AssertN(6, lts.decaytree.size())) {
      A = Regge->ME8(lts);
    } else {
      throw std::invalid_argument(ISTATE + "[" + CHANNEL + "] requires 2, 4 or 6-body final state");
    }
    return abs2(A);
  }
};

class PROC_16 : public MProc {
 public:
  PROC_16() : MProc("PP", "CON-", {"Hadron continuum 2-body with [t-u] amp.", "Pomeron"}) {}
  ~PROC_16() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitRegge(lts);

    if (!AssertN(2, lts.decaytree.size())) {
      throw std::invalid_argument(ISTATE + "[" + CHANNEL + "] requires 2-body final state");
    }
    std::complex<double> A = Regge->ME4(lts, -1);
    return abs2(A);
  }
};


class PROC_17 : public MProc {
 public:
  PROC_17() : MProc("PP", "RES+CON", {"Hadron resonances + continuum 2-body", "Pomeron / yP"}) {}
  ~PROC_17() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitRegge(lts);
    std::complex<double> A = 0.0;

    if (!AssertN(2, lts.decaytree.size())) {
      throw std::invalid_argument(ISTATE + "[" + CHANNEL + "] requires 2-body final state");
    }

    // 1. Continuum matrix element
    A = Regge->ME4(lts, 1);

    // 2. Coherent sum of Resonances (loop over)
    for (auto& x : lts.RESONANCES) {
```



```
      const int J = static_cast<int>(x.second.p.spinX2 / 2.0);

      // Gamma-Pomeron for vectors
      if (J == 1 && x.second.p.P == -1) {
        A += Regge->PhotoME3(lts, x.second);
      }
      // Pomeron-Pomeron, J = 0,1,2,... all ok
      else {
        A += Regge->ME3(lts, x.second);
      }
    }

    // ----------------------------------------------------------------
    // Set helicity amplitudes for the screening loop
    lts.hamp = {A};
    // ----------------------------------------------------------------

    return abs2(A);
  }
};

// ------------------------------------------------------------------------
// Gamma(Odderon)-Pomeron processes
// ------------------------------------------------------------------------

class PROC_18 : public MProc {
 public:
  PROC_18() : MProc("yP", "RES", {"Photoproduced resonance", "kt-EPA x Pomeron"}) {}
  ~PROC_18() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitRegge(lts);
    std::complex<double> A = 0.0;

    // Coherent sum of Resonances (loop over)
    for (auto& x : lts.RESONANCES) {
      const int J = static_cast<int>(x.second.p.spinX2 / 2.0);
      // Vectors only
      if (J == 1 && x.second.p.P == -1) {
        A += Regge->PhotoME3(lts, lts.RESONANCES.begin()->second);
      }
    }
    return abs2(A);  // amplitude squared
  }
};

class PROC_19 : public MProc {
 public:
  PROC_19() : MProc("yP", "RESTENSOR", {"Photoproduced resonance", "QED x Tensor Pomeron"}) {}
  ~PROC_19() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitTensor(lts);
    return Tensor->ME3(lts);
  }
};


class PROC_20 : public MProc {
 public:
  PROC_20() : MProc("OP", "RES", {"Oddproduced resonance", "Odderon x Pomeron"}) {}
  ~PROC_20() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
    InitRegge(lts);
    std::complex<double> A = 0.0;
    // Coherent sum of Resonances (loop over)
    for (auto& x : lts.RESONANCES) {
      const int J = static_cast<int>(x.second.p.spinX2 / 2.0);
      // Vectors only
      if (J == 1 && x.second.p.P == -1) { A += Regge->ME3ODD(lts, x.second); }
    }
    return abs2(A);  // amplitude squared
  }
};


// ------------------------------------------------------------------------
// Durham QCD processes
// ------------------------------------------------------------------------

class PROC_21 : public MProc {
 public:
  PROC_21() : MProc("gg", "chic(0)", {"QCD resonance chic(0)", "Durham QCD"}) {}
  ~PROC_21() {}
  virtual double Amp2(gra::LORENTZSCALAR& lts) {
```

```
499:      InitDurham(lts);
500:      double amp2 = 0.0;
501:      amp2        = Durham->DurhamQCD(lts, CHANNEL);
502:      return amp2;
503:    }
504: };
505:
506: class PROC_22 : public MProc {
507:  public:
508:   PROC_22()
509:       : MProc("gg", "CON",
510:               {"QCD continuum gg, 2 x pseudoscalar", "Durham QCD", "UNDER VALIDATION!"}) {}
511:   ~PROC_22() {}
512:   virtual double Amp2(gra::LORENTZSCALAR& lts) {
513:     InitDurham(lts);
514:
515:     double amp2 = 0.0;
516:     if (AssertN({21, 21), PDGlist(lts))) {
517:       amp2 = Durham->DurhamQCD(lts, "gg");
518:     } else if (AssertN(2, lts.decaytree.size())) {
519:       amp2 = Durham->DurhamQCD(lts, "MMbar");
520:     } else {
521:       ThrowUnknownFinalState();
522:     }
523:     return amp2;
524:   }
525: };
526:
527: class PROC_23 : public MProc {
528:  public:
529:   PROC_23()
530:       : MProc("gg", "FLUX", {"Durham flux with |A|^2 = 1", "Durham QCD", "SYSTEM TEST PROCESS!"}) {}
531:   ~PROC_23() {}
532:   virtual double Amp2(gra::LORENTZSCALAR& lts) {
533:     InitDurham(lts);
534:
535:     return Durham->DurhamQCD(lts, "FLUX");
536:   }
537: };
538:
539:
540: // ------------------------------------------------------------------------
541: // Gamma-Gamma processes
542: // ------------------------------------------------------------------------
543:
544: class PROC_24 : public MProc {
545:  public:
546:   PROC_24()
547:       : MProc("yy_LUX", "CON", {"Continuum l+l, qqbar, W+W-, monopolepair", "Collinear LUX-PDF"}) {}
548:   ~PROC_24() {}
549:   virtual double Amp2(gra::LORENTZSCALAR& lts) {
550:     InitGamma(lts);
551:     double amp2 = GammaGammaCON(lts);
552:     return flux::ApplyLUXfluxes(amp2, lts);
553:   }
554: };
555:
556: class PROC_25 : public MProc {
557:  public:
558:   PROC_25()
559:       : MProc("yy_DZ", "CON",
560:               {"Continuum l+l, qqbar, W+W-, monopolepair", "Collinear Drees-Zeppenfeld EPA"}) {}
561:   ~PROC_25() {}
562:   virtual double Amp2(gra::LORENTZSCALAR& lts) {
563:     InitGamma(lts);
564:     double amp2 = GammaGammaCON(lts);
565:     return flux::ApplyDZfluxes(amp2, lts);
566:   }
567: };
568:
569: class PROC_26 : public MProc {
570:  public:
571:   PROC_26()
572:       : MProc("yy", "FLUX", {"kt-EPA flux with |A|^2 = 1", "kt-EPA", "SYSTEM TEST PROCESS!"}) {}
573:   ~PROC_26() {}
574:   virtual double Amp2(gra::LORENTZSCALAR& lts) {
575:     InitGamma(lts);
576:     double amp2 = 1.0;
577:     return flux::ApplyktEPAfluxes(amp2, lts);
578:   }
579: };
580:
581: class PROC_27 : public MProc {
```



```
582:  public:
583:   PROC_27()
584:       : MProc(
585:                 "yy_DZ", "FLUX",
586:                 {"DZ flux with |A|^2 = 1", "Collinear Drees-Zeppenfeld EPA", "SYSTEM TEST PROCESS!"}) {}
587:   ~PROC_27() {}
588:   virtual double Amp2(gra::LORENTZSCALAR& lts) {
589:     InitGamma(lts);
590:     double amp2 = 1.0;
591:     return flux::ApplyDZfluxes(amp2, lts);
592:   }
593: };
594:
595:
596: // Umbrella class
597: class MSubProc {
598:  public:
599:   MSubProc(const std::vector<std::string>& istate, const std::string& mc);
600:   void Initialize(const std::string& istate, const std::string& channel);
601:
602:   MSubProc() {}
603:   ~MSubProc();
604:
605:   double GetBareAmplitude2(gra::LORENTZSCALAR& lts);
606:
607:   std::string  ISTATE;       // "PP","yy","gg" etc.
608:   std::string  CHANNEL;      // "CON", "RES" etc.
609:   unsigned int LIPSDIM = 0;  // Lorentz Invariant Phase Space Dimension
610:
611:   // Process descriptions
612:   std::map<std::string, std::vector<std::string>> Processes;
613:
614:   bool                ProcessExist(std::string process) const;
615:   std::vector<std::string> PrintProcesses() const;
616:   std::vector<std::string> GetProcessDescriptor(std::string process) const;
617:
618:  private:
619:   void ConstructDescriptions(const std::string& istate, const std::string& mc);
620:   void CreateProcesses();
621:   void DeleteProcesses();
622:   void ActivateProcess();
623:
624:   std::vector<MProc*> pr;
625: };
626:
627: }  // namespace gra
628:
629: #endif
```



```
1: // Factorized 2->3 phase space class
2: //
3: // (c) 2017-2020 Mikael Mieskolainen
4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
5:
6: #ifndef MFACTORIZED_H
7: #define MFACTORIZED_H
8:
9: // C++
10: #include <complex>
11: #include <random>
12: #include <vector>
13:
14: // Own
15: #include "Graniitti/M4Vec.h"
16: #include "Graniitti/MAux.h"
17: #include "Graniitti/MMatrix.h"
18: #include "Graniitti/MProcess.h"
19: #include "Graniitti/MSpin.h"
20:
21: // HepMC33
22: #include "HepMC3/GenEvent.h"
23:
24: namespace gra {
25: class MFactorized : public MProcess {
26:  public:
27:   MFactorized();
28:   MFactorized(std::string process, const std::vector<aux::OneCMD> &syntax);
29:   virtual ~MFactorized();
30:
31:   void post_Constructor();
32:
33:   double operator()(const std::vector<double> &randvec, AuxIntData &aux) {
34:     return EventWeight(randvec, aux);
35:   }
36:   double EventWeight(const std::vector<double> &randvec, AuxIntData &aux);
37:   bool   EventRecord(HepMC3::GenEvent &evt);
38:   void   PrintInit(bool silent) const;
39:
40:  private:
41:   void Initialize();
42:   bool LoopKinematics(const std::vector<double> &p1p, const std::vector<double> &p2p);
43:   bool FiducialCuts() const;
44:
45:   // 5+1-Dim phase space, 2->3
46:   bool B51RandomKin(const std::vector<double> &randvec);
47:   bool B51BuildKin(double pt1, double pt2, double phi1, double phi2, double yX, double m2X,
48:                    double m1, double m2);
49:   void B51RecordEvent(HepMC3::GenEvent &evt);
50:
51:   double B51IntegralVolume() const;
52:   double B51PhaseSpaceWeight() const;
53:
54:   void DecayWidthPS(double &exact) const;
55:
56:   // Dynamic sampling boundaries based on resonance position and width
57:   double M_MIN = 0.0;
58:   double M_MAX = 0.0;
59: };
60:
61: }  // namespace gra
62:
63: #endif
```



```
1: // Standard relativistic kinematics and related structures [HEADER ONLY file]
2: //
3: // (c) 2017-2020 Mikael Mieskolainen
4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
5:
6: #ifndef MKINEMATICS_H
7: #define MKINEMATICS_H
8:
9:
10: // C++
11: #include <complex>
12: #include <random>
13: #include <valarray>
14: #include <vector>
15:
16: // Own
17: #include "Graniitti/M4Vec.h"
18: #include "Graniitti/MAux.h"
19: #include "Graniitti/MCW.h"
20: #include "Graniitti/MMatOper.h"
21: #include "Graniitti/MMath.h"
22: #include "Graniitti/MMatrix.h"
23: #include "Graniitti/MPDG.h"
24: #include "Graniitti/MParticle.h"
25: #include "Graniitti/MResonance.h"
26: #include "Graniitti/MSudakov.h"
27:
28:
29: namespace gra {
30: namespace kinematics {
31:
32: // Case m3 = m4
33: inline double SolvePz3_A(double m3, double pt3, double pt4, double pz5, double E5, double s) {
34:   const double t2  = E5 * E5;
35:   const double t3  = pt3 * pt3;
36:   const double t4  = pt4 * pt4;
37:   const double t5  = pz5 * pz5;
38:   const double t6  = m3 * m3;
39:   const double t7  = std::sqrt(s);
40:   const double t8  = s * t2 * 6.0;
41:   const double t9  = std::pow(s, 3.0 / 2.0);
42:   const double t10 = t2 * t2;
43:   const double t11 = t3 * t3;
44:   const double t12 = t4 * t4;
45:   const double t13 = t5 * t5;
46:   const double t14 = s * s;
47:   const double t15 = t5 * t6 * 4.0;
48:   const double t16 = t3 * t5 * 2.0;
49:   const double t17 = t4 * t5 * 2.0;
50:   const double t18 = E5 * t6 * t7 * 8.0;
51:   const double t19 = E5 * t3 * t7 * 4.0;
52:   const double t20 = E5 * t4 * t7 * 4.0;
53:   const double t21 = E5 * t5 * t7 * 4.0;
54:   const double t22 = t8 + t10 + t11 + t12 + t13 + t14 + t15 + t16 + t17 + t18 + t19 + t20 + t21 -
55:                      E5 * t9 * 4.0 - s * t3 * 2.0 - s * t4 * 2.0 - s * t5 * 2.0 - s * t6 * 4.0 -
56:                      t2 * t3 * 2.0 - t2 * t4 * 2.0 - t2 * t5 * 2.0 - t3 * t4 * 2.0 - t2 * t6 * 4.0 -
57:                      E5 * t2 * t7 * 4.0;
58:   const double t23 = std::sqrt(t22);
59:   const double t0  = pz5 * (-1.0 / 2.0) - (E5 * t23 * (1.0 / 2.0) - t7 * t23 * (1.0 / 2.0) +
60:                                            pz5 * (t3 * (1.0 / 2.0) - t4 * (1.0 / 2.0))) /
61:                                               (s + t2 - t5 - E5 * t7 * 2.0);
62:
63:   return t0;
64: }
65:
66: /*
67: // Old version for reference
68: inline double SolvePz3_B(double m1, double m2, double pt1, double pt2, double
69: pz, double pE, double
70: s) {
71:
72:   using gra::math::msqrt;
73:   using gra::math::pow2;
74:   using gra::math::pow3;
75:   using gra::math::pow4;
76:   using gra::math::pow5;
77:
78:   const double sqrt_s = msqrt(s);
79:   const double piz =
80:             (-pow2(m1) * pow2(pE) * pz) + pow2(m2) * pow2(pE) * pz -
81:              pow4(pE) * pz - pow2(pE) * pow2(pt1) * pz +
82:              pow2(pE) * pow2(pt2) * pz + pow2(m1) * pow3(pz) -
83:              pow2(m2) * pow3(pz) + 2.0 * pow2(pE) * pow3(pz) +
```

```
            pow2(pt1) * pow3(pz) - pow2(pt2) * pow3(pz) - pow5(pz) -
            2.0 * pow2(m1) * pE * pz * sqrt_s +
            2.0 * pow2(m2) * pE * pz * sqrt_s -
            2.0 * pE * pow2(pt1) * pz * sqrt_s +
            2.0 * pE * pow2(pt2) * pz * sqrt_s - pow2(m1) * pz * s +
            pow2(m2) * pz * s + 2.0 * pow2(pE) * pz * s - pow2(pt1) * pz * s +
            pow2(pt2) * pz * s + 2.0 * pow3(pz) * s - pz * pow2(s) +
            sqrt(pow2(pE - sqrt_s) *
                          pow2(pow2(pE) - pow2(pz) + 2.0 * pE * sqrt_s + s) *
                         (pow4(m1) + pow4(m2) + pow4(pE) -
                          2.0 * pow2(pE) * pow2(pt1) + pow4(pt1) -
                          2.0 * pow2(pE) * pow2(pt2) - 2.0 * pow2(pt1) * pow2(pt2) +
                          pow4(pt2) - 2.0 * pow2(pE) * pow2(pz) +
                          2.0 * pow2(pt1) * pow2(pz) + 2.0 * pow2(pt2) * pow2(pz) +
                          pow4(pz) - 4.0 * pow3(pE) * sqrt_s +
                          4.0 * pE * pow2(pt1) * sqrt_s +
                          4.0 * pE * pow2(pt2) * sqrt_s +
                          4.0 * pE * pow2(pz) * sqrt_s + 6.0 * pow2(pE) * s -
                          2.0 * pow2(pt1) * s - 2.0 * pow2(pt2) * s -
                          2.0 * pow2(pz) * s - 4.0 * pE * std::pow(s, 1.5) + pow2(s) -
                          2.0 * pow2(m2) * (pow2(pE) + pow2(pt1) - pow2(pt2) -
                                            pow2(pz) - 2.0 * pE * sqrt_s + s) -
                          2.0 * pow2(m1) *
                                (pow2(m2) + pow2(pE) - pow2(pt1) + pow2(pt2) -
                                 pow2(pz) - 2.0 * pE * sqrt_s + s)))) /
            (2.0 * (pow4(pE) + pow2(pow2(pz) - s) -
                    2.0 * pow2(pE) * (pow2(pz) + s)));

            return plz;
}
*/

// Case m3, m4 same or different
inline double SolvePz3_B(double m3, double m4, double pt3, double pt4, double pz5, double E5,
                         double s) {
  const double t2  = E5 * E5;
  const double t3  = m3 * m3;
  const double t4  = m4 * m4;
  const double t5  = pt3 * pt3;
  const double t6  = pt4 * pt4;
  const double t7  = pz5 * pz5;
  const double t8  = std::sqrt(s);
  const double t9  = s * t2 * 6.0;
  const double t10 = std::pow(s, 3.0 / 2.0);
  const double t11 = t2 * t2;
  const double t12 = t3 * t3;
  const double t13 = t4 * t4;
  const double t14 = t5 * t5;
  const double t15 = t6 * t6;
  const double t16 = t7 * t7;
  const double t17 = s * s;
  const double t18 = t3 * t5 * 2.0;
  const double t19 = t4 * t6 * 2.0;
  const double t20 = t3 * t7 * 2.0;
  const double t21 = t4 * t7 * 2.0;
  const double t22 = t5 * t7 * 2.0;
  const double t23 = t6 * t7 * 2.0;
  const double t24 = E5 * t3 * t8 * 4.0;
  const double t25 = E5 * t4 * t8 * 4.0;
  const double t26 = E5 * t5 * t8 * 4.0;
  const double t27 = E5 * t6 * t8 * 4.0;
  const double t28 = E5 * t7 * t8 * 4.0;
  const double t29 = t9 + t11 + t12 + t13 + t14 + t15 + t16 + t17 + t18 + t19 + t20 + t21 + t22 +
                     t23 + t24 + t25 + t26 + t27 + t28 - E5 * t10 * 4.0 - s * t3 * 2.0 -
                     s * t4 * 2.0 - s * t5 * 2.0 - s * t6 * 2.0 - s * t7 * 2.0 - t2 * t3 * 2.0 -
                     t2 * t4 * 2.0 - t2 * t5 * 2.0 - t3 * t4 * 2.0 - t2 * t6 * 2.0 - t2 * t7 * 2.0 -
                     t3 * t6 * 2.0 - t4 * t5 * 2.0 - t5 * t6 * 2.0 - E5 * t2 * t8 * 4.0;
  const double t30 = std::sqrt(t29);
  const double t0 =
      pz5 * (-1.0 / 2.0) -
      (E5 * t30 * (1.0 / 2.0) - t8 * t30 * (1.0 / 2.0) +
       pz5 * (t3 * (1.0 / 2.0) - t4 * (1.0 / 2.0) + t5 * (1.0 / 2.0) - t6 * (1.0 / 2.0))) /
          (s + t2 - t7 - E5 * t8 * 2.0);

  return t0;
}

/*
%% MATLAB symbolic code to generate solutions
%% Solve the non-linear system
syms E3 E4 E5 pz3 pz4 pz5 m3 m4 pt3 pt4 s
//
//
```



```
// assume(E3  > 0);
// assume(E4  > 0);
// assume(pt3 > 0);
// assume(pt4 > 0);
// assume(m3  > 0);
// assume(m4  > 0);
// assume(s   > 0)
//
% Use CMS-frame
eqs = [0       == pz3 + pz4 + pz5
       s^(1/2) == E3 + E4 + E5
       E3^2    == m3^2 + pz3^2 + pt3^2
       E4^2    == m4^2 + pz4^2 + pt4^2].';

S = solve(eqs, [pz3 pz4 E3 E4]);

%% The answer for p3z, rest by substitution
S = simplify(S.pz3, 25);

polybranch = 2; % Polynomial branch
S = S(polybranch);


str = 'const double';

% Elastic case
sol_A = simplify(subs(S, [m4], [m3]), 'Steps', 25);
ccode(sol_A, 'File', 'temp.c'); readwritetext('temp.c','solution_A.c', str);

% Generic inelastic case
sol_B = simplify(S, 'Steps', 25);
ccode(sol_B, 'File', 'temp.c'); readwritetext('temp.c','solution_B.c', str);
*/

/*
function readwritetext(old_file, new_file, str)
fid_old = fopen(old_file,'r');
fid_new = fopen(new_file,'w');

tline = fgetl(fid_old);
while ischar(tline)
        fprintf(fid_new, '%s%s\n', str, tline);
        tline = fgetl(fid_old);
end
fclose(fid_old);
fclose(fid_new);
end
*/

// This function solves pz3 component (in center-of-momentum frame)
// in p1 + p2 -> p3 + (p5) + p4
//
//
inline double SolvePz(double m3, double m4, double pt3, double pt4, double pz5, double E5,
                      double s) {
  constexpr double EPS = 1e-9;

  if (std::abs(m3 - m4) < EPS) {  // identical mass case (elastic-X-elastic)
    return SolvePz3_A(m3, pt3, pt4, pz5, E5, s);
  } else {
    return SolvePz3_B(m3, m4, pt3, pt4, pz5, E5, s);
  }
}

// Center of mass momentum^2
// for e.g. 1+2 (p_(cm)^2) -> 3+4 (p_(cm)'^2)
//
// dsigma/dOmega (s,theta) = 1/(64*pi^2*s) p_(cm)'/p_(cm) |M|^2,
// where dOmega = dcostheta dphi, if no initial state polarization, no phi dep.
//
// Then integrating phi: dcostheta dphi / (16*pi^2) = dcostheta/(8*pi), gives
// dsigma/dt (s,t) = 1/(16*pi*s^2 * p_(cm)^2) |M(s,t)|^2
//
inline double pcm2(double s, double m1, double m2) {
  return 1 / (4.0 * s) * (s - gra::math::pow2(m1 + m2)) * (s - gra::math::pow2(m1 - m2));
}

// 2->2 scattering invariants (p1 + p2  -> p3 + p4)
//
// -s/2(1 - cos(theta*))
inline double mandelstam_t(const M4Vec &p1, const M4Vec &p3) { return -2.0 * (p1 * p3); }
// -s/2(1 + cos(theta*))
inline double mandelstam_u(const M4Vec &p2, const M4Vec &p4) { return -2.0 * (p2 * p4); }
```



```

// 1-body Lorentz invariant phase space volume integral for energy < MAX (GeV)
// \int dPS_1 = \int d^3p / (2E(2pi)^3)
//            = \int_0^{4\pi} d^2\Omega/(2pi)^3 \int_0^{\sqrt(MAX^2-m^2)} p^2
//              dp/(2E)
//            = 4\pi / (2(2\pi)^3) \sqrt(MAX^2 - m^2)^3/(3 MAX)
//
// with E^2 = p^2 + m^2
//
inline double dPhi1(double MAX, double m) {
  using gra::math::pow2;

  return pow2(MAX) / (12 * pow2(gra::math::PI)) * std::pow(1.0 - pow2(m) / pow2(MAX), 3.0 / 2.0);
}

// Two-body Lorentz invariant phase space volume, evaluated in the rest frame
// p_norm is the result given by DecayMomentum()
inline double dPhi2(double E, double p_norm) {
  // Sampling box volume cos(theta)  [-1, 1], phi  [0,2pi]
  const double V = 2.0 * 2.0 * gra::math::PI;
  return V * p_norm / (16.0 * gra::math::pow2(gra::math::PI) * E);
}

// Kallen triangle function (square root applied here)
// symmetric under interchange of x <-> y <-> z:
// lambda^(1/2) = ( (x - y - z)^2 - 4yz )^(1/2)
//
// 1 / lambda^(1/2)(s,m_a^2,m_b^2) -> 1/(2s) when s >> m_a^2, m_b^2
//
// Note x,y,z are Lorentz scalars (ENERGY squared variables)
inline double SqrtKallenLambda(double x, double y, double z) {
  // Hyperboloid formula
  return gra::math::msqrt(x * x + y * y + z * z - 2.0 * (x * y + x * z + y * z));
  //     return msqrt( pow2( x - y - z) - 4.0*y*z ); // numerically unstable
}

inline double beta12(double s, double m1, double m2) {
  return SqrtKallenLambda(s, gra::math::pow2(m1), gra::math::pow2(m2)) / s;
}

// Kinematic (momentum) bound for phase space (decays)
// This is same as: SqrtKallenLambda(x^2, y^2, z^2) / (2x)
//
// Note x,y,z [mother, daughter1, daughter2] in units of ENERGY (not ENERGY^2)
inline double DecayMomentum(double x, double y, double z) {
  // return SqrtKallenLambda(x*x, y*y, z*z) / (2*x);
  // numerically stable
  return gra::math::msqrt((x - y - z) * (x + y + z) * (x + y - z) * (x - y + z)) / (2.0 * x);
}

// Massive two body phase space integral volume PS^(2)(M^2; m_A^2, m_B^2)
inline double PS2Massive(double M2, double mA2, double mB2) {
  return SqrtKallenLambda(M2, mA2, mB2) / (8.0 * gra::math::PI * M2);
}

// Massless phase space integral volume PS^((n)) = \int dPS^((n))
// where M2 (GeV^2) is the total CMS energy^2 of the n-body system.
// Use this e.g. as a (debug) reference of other routines
// Gives 1/(8*PI) for n = 2 case, regardless of M^2
//
// tgamma is C++/11 math Gamma function
inline double PSnMassless(double M2, int n) {
  // const double denom = tgamma(n) * tgamma(n - 1.0);
  const double denom = gra::math::factorial(n - 1) * gra::math::factorial(n - 2);
  const double E     = gra::math::msqrt(M2);
  const double PS =
      (1.0 / (2.0 * std::pow(4.0 * gra::math::PI, 2 * n - 3))) * (std::pow(E, 2 * n - 4) / denom);

  return PS;
}

// Exact 2-body partial decay width,
// is not dependent on momentum => phase space and matrix element factorize
//
// Input: Mother mass M0^2
//        Decay product 1 mass^2
//        Decay product 2 mass^2
//        Decay matrix element squared |M|^2 value
//        Final state symmetry factor S
//
// [REFERENCE: Alwall et al, https://arxiv.org/abs/1402.1178]
inline double PDW2body(double M0_2, double m1_2, double m2_2, double MatElem2, double S_factor) {
  return SqrtKallenLambda(M0_2, m1_2, m2_2) * MatElem2 /
```



```
         (16.0 * gra::math::PI * S_factor * math::pow3(math::msqrt(M0_2)));
}

// Relativistic velocity of a particle with m^2
// in the system with invariant shat
inline double Beta(double m2, double shat) { return gra::math::msqrt(1.0 - 4.0 * m2 / shat); }

// 2->2 Scattering E1 + E2 -> E3 + E4 angle as a function of (s,t,m_i^2)
// [REFERENCE: Kallen, Elementary Particle Physics, Addison-Wesley, 1964]
inline double costhetastar(double s, double t, double E1, double E2, double E3, double E4) {
  const double numer = s * s + s * (2.0 * t - E1 - E2 - E3 - E4) + (E1 - E2) * (E3 - E4);
  const double denom = SqrtKallenLambda(s, E1, E2) * SqrtKallenLambda(s, E3, E4);

  return numer / denom;
}

// 2->2 Scattering E1 + E2 -> E3 + E4 Mandelstam t-limits (physical boundaries)
// [REFERENCE: Kallen, Elementary Particle Physics, Addison-Wesley, 1964]
// [REFERENCE: Sjostrand, Mrenna, Skands, https://arxiv.org/abs/hep-ph/0603175]
inline void Two2TwoLimit(double s, double E1, double E2, double E3, double E4, double &tmin,
                         double &tmax) {
  // Triangle function
  const double KL12 = SqrtKallenLambda(s, E1, E2);
  const double KL34 = SqrtKallenLambda(s, E3, E4);

  // Geometry
  const double A = s - (E1 + E2 + E3 + E4) + (E1 - E2) * (E3 - E4) / s;
  const double B = KL12 * KL34 / s;
  const double C = (E3 - E1) * (E4 - E2) + (E1 + E4 - E2 - E3) * (E1 * E4 - E2 * E3) / s;
  tmin           = -0.5 * (A + B);
  tmax           = C / tmin;
}

// Standard conventions:
//
// E^2 = (mc^2)^2 + |pc|^2 = gamma*mc^2, p = gamma*m*v, m = constant!
// gamma = E/m = 1/sqrt(1-v^2/c^2) = 1/sqrt(1-beta^2)
// beta  = v/c = p/E => gamma*beta = p/m
//
//
// d^3p / (2E(2pi)^3) = |\partial(p_x,p_y,p_z)/\partial(p_t,\phi,y)| dp_t d\phi
//   dy / (2E(2pi)^3) = p_t^3 d_pt d\phi dy / (2E^2(2\pi)^3)
//
//

// Generic Lorentz boost into the direction given by 4-momentum of "boost"
//
// A. sign = -1 for boost to the rest frame (out of lab, for example)
// B. sign =  1 out from the rest frame (back to lab, for example)
//
template <typename T>
inline void LorentzBoost(const T &boost, double M0, T &p, int sign) {
  std::valarray<double> p3 = {p.Px(), p.Py(), p.Pz()};

  // Mother beta-vector
  const std::valarray<double> beta = {sign * boost.Px() / boost.E(), sign * boost.Py() / boost.E(),
                                      sign * boost.Pz() / boost.E()};

  // Mother gamma-factor
  const double gamma = boost.E() / M0;

  // Apply transforms
  const double kappa1 = (beta * p3).sum();  // Inner product
  const double kappa2 = gamma * (gamma * kappa1 / (1.0 + gamma) + p.E());

  // Vector sum
  p3 += kappa2 * beta;

  p = T(p3[0], p3[1], p3[2], gamma * (p.E() + kappa1));
}

// Flat variables in spherical coordinates
template <typename T2>
inline void FlatIsotropic(double &costheta, double &sintheta, double &phi, T2 &rng) {
  costheta = rng.U(-1.0, 1.0);  // cos(theta) flat [-1,1]
  sintheta = gra::math::msqrt(1.0 - gra::math::pow2(costheta));
  phi      = rng.U(0.0, 2.0 * gra::math::PI);  // phi flat [0,2pi]
}

// Isotropic decay 0 -> 1 + 2 in spherical coordinates with flat cos(theta),phi
template <typename T1, typename T2>
inline void Isotropic(double pnorm, T1 &p1, T1 &p2, double m1, double m2, T2 &rng) {
  using gra::math::msqrt;
  using gra::math::pow2;
```

```
416:
417:   double costheta, sintheta, phi;
418:   FlatIsotropic(costheta, sintheta, phi, rng);
419:
420:   // Jacobian of spherical coordinates
421:   const std::valarray<double> k = {pnorm * sintheta * std::cos(phi),
422:                                    pnorm * sintheta * std::sin(phi), pnorm * costheta};
423:
424:   // Energies by on-shell condition
425:   const std::valarray<double> e = {msqrt(pow2(m1) + pow2(pnorm)), msqrt(pow2(m2) + pow2(pnorm))};
426:
427:   // Back-to-back
428:   p1 = T1(k[0], k[1], k[2], e[0]);
429:   p2 = T1(-k[0], -k[1], -k[2], e[1]);
430: }
431:
432: // 2-Body isotropic phase space decay in the rest frame
433: //
434: //           m0
435: //          /
436: //         /
437: // M0 ----*----- m1
438: //
439: // For floating point / efficiency reasons, the mother mass needs to be provided
440: // outside
441: //
442: template <typename T1, typename T2>
443: inline MCW TwoBodyPhaseSpace(const T1 &mother, double M0, const std::vector<double> &m,
444:                              std::vector<T1> &p, T2 &rng) {
445:   // Two particles
446:   p.resize(2);
447:
448:   // First isotropic decay
449:   const double pnorm = DecayMomentum(M0, m[0], m[1]);
450:   Isotropic(pnorm, p[0], p[1], m[0], m[1], rng);
451:
452:   // Boost daughters to the lab frame
453:   const int sign = 1;
454:   for (const auto &i : {0, 1}) { LorentzBoost(mother, M0, p[i], sign); }
455:
456:   // return phase space weight
457:   const double weight = dPhi2(M0, pnorm);
458:   return MCW(weight, gra::math::pow2(weight), 1);
459: }
460:
461: // 3-Body isotropic (Dalitz) phase space decay in the rest frame;
462: // by factorizing the 3-body phase space into two 2-body
463: //
464: //           m[0]       m[1]
465: //            /          /
466: //           /          /
467: // M0 ----*---m12---*----- m[2]
468: //
469: // dphi_3(pM; p0, p1, p2) = dM_12^2 dphi_2 (pM; p0, p12) dphi_2 (p12; p1, p2)
470: //
471: // Return: phase space weight for a valid fragmentation and -1.0 for a
472: // kinematically impossible.
473: // When unweight == true, This function returns also number of trials for proper
474: // MC error treatment
475: // (MCW)
476: //
477: template <typename T1, typename T2>
478: inline MCW ThreeBodyPhaseSpace(const T1 &mother, double M0, const std::vector<double> &m,
479:                                std::vector<T1> &p, bool unweight, T2 &rng) {
480:   p.resize(3);  // Three final states
481:   T1 p12;
482:
483:   // Phase space boundaries [min,max]
484:   const std::vector<double> m12bound = {m[1] + m[2], M0 - m[0]};
485:   double w_max = DecayMomentum(M0, m[0], m12bound[0]) * DecayMomentum(m12bound[1], m[1], m[2]);
486:   if (unweight == false) { w_max = 0; }
487:
488:   // Accceptance-Rejection
489:   const unsigned int  MAXTRIAL = 1e8;
490:   std::vector<double> pnorm    = {0.0, 0.0};
491:   double              m12      = 0;
492:   MCW                 w;
493:   do {
494:     m12      = rng.U(m12bound[0], m12bound[1]);  // Flat mass (in GeV, not GeV^2)
495:     pnorm[0] = DecayMomentum(M0, m[0], m12);
496:     pnorm[1] = DecayMomentum(m12, m[1], m[2]);
497:
498:     const double w = (m12 / gra::math::PI) * dPhi2(M0, pnorm[0]) * dPhi2(m12, pnorm[1]);
```



```
499:     x.Push(w);
500:     if (x.GetN() > MAXTRIAL) { return MCW(-1, 0, 0); }  // Impossible kinematics
501:   } while ((pnorm[0] * pnorm[1]) < rng.U(0.0, w_max));
502:
503:   // pM -> p0 + p12 in the pM r.f. and p12 -> p1 + p2 in the p12 r.f.
504:   Isotropic(pnorm[0], p[0], p12, m[0], m12, rng);
505:   Isotropic(pnorm[1], p[1], p[2], m[1], m[2], rng);
506:
507:   // Boost p[1] and p[2] out from the p12 rest frame to the pM rest frame
508:   const int sign = 1;  // Boost sign
509:   for (const auto &i : {1, 2}) { LorentzBoost(p12, m12, p[i], sign); }
510:
511:   // Boost p[0], p[1] and p[2] to the lab frame
512:   for (const auto &i : {0, 1, 2}) { LorentzBoost(mother, M0, p[i], sign); }
513:
514:   // Phasespace weightÂ [note m versus m^2 jacobian, taken into account]
515:   // [m12 volume (GeV)] x [dm_12] x [dphi2] x [dphi2]
516:   const double volume = (m12bound[1] - m12bound[0]);
517:   x                   = x * volume;  // operator overloaded
518:
519:   // Return phase space weight
520:   return x;
521: }
522:
523: // Decay setup for NBodyPhaseSpace
524: // Uses the "sorting algorithm", see description in F. James, CERN/68
525: // bool unweighted for (un)weighted operation
526: //
527: // Return: W for a valid and -1.0 for a kinematically impossible
528: //
529: template <typename T1, typename T2>
530: inline MCW NBodySetup(const T1 &mother, double M0, const std::vector<double> &m,
531:                       std::vector<double> &M_eff, std::vector<double> &pnorm, bool unweight,
532:                       T2 &rng) {
533:   // Decay multiplicity (1 -> N decay)
534:   const unsigned int  N = m.size();
535:   std::vector<double> randvec(N - 2, 0.0);
536:
537:   // Random variables functor
538:   auto fillrandom = [&](double &value) -> void { value = rng.U(0, 1); };
539:
540:   // Effective (intermediate) masses
541:   M_eff[0]     = m[0];  // First daughter
542:   M_eff[N - 1] = M0;    // Mother
543:
544:   // Cumulative sum of daughter masses
545:   std::vector<double> sumvec;
546:   gra::math::CumSum(m, sumvec);
547:
548:   // Sampling mass interval
549:   const double DELTA = M_eff[N - 1] - sumvec[N - 1];
550:
551:   // Generate effective masses functor, where sumvec[i] defines the minimum
552:   auto calcmass = [&]() -> void {
553:     for (std::size_t i = 1; i < N - 1; ++i) { M_eff[i] = sumvec[i] + randvec[i - 1] * DELTA; }
554:   };
555:
556:   // ------------------------------------------------------------------
557:   // Recursively find maximum weight for Acceptance-Rejection
558:   double              w_max   = 1.0;
559:   std::vector<double> m_minmax = {0.0, DELTA + m[N - 1]};
560:   for (std::size_t i = 1; i < N; ++i) {
561:     m_minmax[0] += m[i - 1];
562:     m_minmax[1] += m[i];
563:     w_max *= DecayMomentum(m_minmax[1], m_minmax[0], m[i]);
564:   }
565:
566:   // Functor to calculate decay momentum to pnorm[], and return product
567:   auto calcmomentum = [&]() -> double {
568:     double w = 1.0;
569:     for (std::size_t i = 1; i < N; ++i) {
570:       pnorm[i] = DecayMomentum(M_eff[i], M_eff[i - 1], m[i]);
571:       w *= pnorm[i];
572:     }
573:     return w;
574:   };
575:
576:   // ------------------------------------------------------------------
577:   // Acceptance-Rejection
578:
579:   if (unweight == false) { w_max = 0; }
580:
581:   const unsigned int MAXTRIAL = 1e8;
```



```
582:   MCW                xj;
583:   double             w = 0.0;
584:   do {
585:     // 1. Ordered random numbers
586:     std::for_each(randvec.begin(), randvec.end(), fillrandom);
587:     std::sort(randvec.begin(), randvec.end());  // Ascending
588:     // std::sort(randvec.begin(), randvec.end(), [](const double a, const double
589:     // b) {
590:     // return a > b; }); // Descending
591:
592:     // 2. Calculate effective masses
593:     calcmass();
594:
595:     // 3. Calculate momentum norm values
596:     w = calcmomentum();
597:
598:     x.Push(w);
599:     if (x.GetN() > MAXTRIAL) { return MCW(-1, 0, 0); }  // Impossible kinematics
600:   } while (w < rng.U(0.0, w_max));
601:
602:   // Normalize the phase space integral
603:   const double volume = 1.0 / (2.0 * std::pow(2.0 * gra::math::PI, 2 * N - 3)) *
604:                         std::pow(DELTA, N - 2) / gra::math::factorial(N - 2) / M0;
605:   x = x * volume;  // operator overloaded
606:
607:   // Return phase space weight
608:   return x;
609: }
610:
611: // Flat N-body phase space by implementing a number of (N-2) recursive 2-body
612: // decays.
613: // Input as [mother 4-vector, vector of daughter masses, vector of final
614: // 4-vectors]
615: //
616: // [REFERENCE: F. James, https://cds.cern.ch/record/275743/files/CERN-68-15.pdf]
617: //
618: //                             m_n
619: //                          *            m_(n-1)
620: //  N-bodies                *           *
621: //          ---------------- *          *          m_(n-2)
622: //        ----------------A------------ *         *
623: //  M_N  ------------------------------B------------ *
624: //       ---------------------------------------------C
625: //                        M_(N-1)      M_(N-2)        *
626: //                                                     *
627: //                                                      *
628: //                                                       M_(N-3)
629: //
630: // For faster alternatives, investigate:
631: // M.M. Block, Monte Carlo phase space evaluation, Comp. Phys. Commun.
632: // 69, 459 (1992)
633: // S. Platzer, RAMBO on diet, https://arxiv.org/abs/1308.2922
634: //
635: // Return: weight for a valid fragmentation and -1.0 for a kinematically
636: // impossible
637: //
638: // When unweight == true, This function returns also number of trials for proper
639: // MC error treatment
640: // (MCW obj)
641: //
642: template <typename T1, typename T2>
643: inline MCW NBodyPhaseSpace(const T1 &mother, double M0, const std::vector<double> &m,
644:                            std::vector<T1> &p, bool unweight, T2 &rng) {
645:   using gra::math::msqrt;
646:   using gra::math::pow2;
647:
648:   const int N = m.size();
649:
650:   // Generate effective masses and decay momentum
651:   std::vector<double> M_eff(N, 0.0);  // Effective masses
652:   std::vector<double> pnorm(N, 0.0);  // Decay momentum weights
653:   const MCW           x = NBodySetup(mother, M0, m, M_eff, pnorm, unweight, rng);
654:   if (x.GetW() < 0) { return x; }  // Impossible kinematics
655:
656:   // Setup decay daughters size
657:   p.resize(N);
658:
659:   // Recursively go down from N-body to 1-particle
660:   T1 k = mother;
661:   for (std::size_t i = N - 1; i >= 1; --i) {
662:     double costheta, sintheta, phi;
663:     FlatIsotropic(costheta, sintheta, phi, rng);
664:     const double pt = pnorm[i] * sintheta;
```



```
665:
666:     p[i] = T1(pt * std::cos(phi), pt * std::sin(phi), pnorm[i] * costheta,
667:               msqrt(pow2(m[i]) + pow2(pnorm[i])));
668:
669:     // Boost to the lab frame
670:     const int sign = 1;  // positive
671:     LorentzBoost(k, k.M(), p[i], sign);
672:
673:     // Subtract the momentum
674:     k -= p[i];
675:   }
676:   // Finally, we are left with only one daughter
677:   p[0] = k;
678:
679:   return x;  // Phase-space weight
680: }
681:
682: // SO(3) Rotations
683:
684: // Rotation matrix from 3-vector a to 3-vector b (both need to be unit vectors with ||x|| = 1)
685: //
686: inline MMatrix<double> RotOnetoOne(const std::vector<double> &a, const std::vector<double> &b) {
687:   if (a.size() != 3 || b.size() != 3) {
688:     throw std::invalid_argument("kinematics::RotOnetoOne: Input should be 3-vectors");
689:   }
690:   // Dot product
691:   const double c = matoper::VecVecMultiply(a, b);
692:
693:   // Cross product
694:   const std::vector<double> v = matoper::Cross(a, b);
695:
696:   // Skew symmetric cross product matrix
697:   const MMatrix<double> Vx = {{0, -v[2], v[1]}, {v[2], 0, -v[0]}, {-v[1], v[0], 0}};
698:   // Identity
699:   const MMatrix<double> I(3, 3, "eye");
700:
701:   // Rotation matrix, with singularity at c = -1 (a and b back-to-back)
702:   const MMatrix<double> R = I + Vx + (Vx * Vx) * (1.0 / (1.0 + c));
703:
704:   return R;
705: }
706:
707: // Rotate 4-vector spatial part by (theta,phi)
708: template <typename T>
709: inline void Rotate(T &p, double theta, double phi) {
710:   const double c1 = std::cos(theta);
711:   const double s1 = std::sin(theta);
712:   const double c2 = std::cos(phi);
713:   const double s2 = std::sin(phi);
714:
715:   // 3x3 Rotation matrix
716:   const MMatrix<double> R = {{c1 * c2, -s2, s1 * c2}, {c1 * s2, c2, s1 * s2}, {-s1, 0.0, c1}};
717:
718:   // Rotate
719:   const std::vector<double> p3    = {p.Px(), p.Py(), p.Pz()};
720:   const std::vector<double> p3new = R * p3;
721:
722:   p = T(p3new[0], p3new[1], p3new[2], p.E());
723: }
724:
725: // Active SO3-rotation counterclockwise around x-axis
726: template <typename T>
727: inline void RotateX(T &p, double angle) {
728:   const double costh = std::cos(angle);
729:   const double sinth = std::sin(angle);
730:
731:   // Rotation matrix applied
732:   const double px = p.Px();
733:   const double py = costh * p.Py() - sinth * p.Pz();
734:   const double pz = sinth * p.Py() + costh * p.Pz();
735:
736:   p = T(px, py, pz, p.E());
737: }
738:
739: // Active SO3-rotation counterclockwise around y-axis
740: template <typename T>
741: inline void RotateY(T &p, double angle) {
742:   const double costh = std::cos(angle);
743:   const double sinth = std::sin(angle);
744:
745:   // Rotation matrix applied
746:   const double px = costh * p.Px() + sinth * p.Pz();
747:   const double py = p.Py();
```

```
748:   const double pz = -sinth * p.Px() + costh * p.Pz();
749:
750:   p = T(px, py, pz, p.E());
751: }
752:
753: // Active SO3-rotation counterclockwise around z-axis
754: template <typename T>
755: inline void RotateZ(T &p, double angle) {
756:   const double costh = std::cos(angle);
757:   const double sinth = std::sin(angle);
758:
759:   // Rotation matrix applied
760:   const double px = costh * p.Px() - sinth * p.Py();
761:   const double py = sinth * p.Px() + costh * p.Py();
762:   const double pz = p.Pz();
763:
764:   p = T(px, py, pz, p.E());
765: }
766:
767:
768: // Find the closest 4-vector on lightcone
769: //
770: // Solution by Lagrange multipliers \Nabla f = \lambda \Nabla g
771: //
772: // eq(1) = 2*(px-px0) == -2*lambda*px
773: // eq(2) = 2*(py-py0) == -2*lambda*py
774: // eq(3) = 2*(pz-pz0) == -2*lambda*pz
775: // eq(4) = 2*(E-E0)   ==  2*lambda*E
776: // eq(5) = E^2        == (px^2 + py^2 + pz^2)
777: //
778: // S = solve(eq, [E,px,py,pz,lambda])
779: //
780: inline M4Vec LagrangeLightCone(const M4Vec &p0) {
781:   // Spacelike (q^2 < 0) or timelike (q^2 > 0) input
782:   const double sign = p0.M2() <= 0 ? 1.0 : -1.0;
783:
784:   const double p3mod2 = p0.P3mod2();
785:   const double p3mod  = math::msqrt(p3mod2);
786:
787:   const double E     = p0.E() / 2 + sign * p3mod / 2;
788:   const double alpha = (p3mod2 + sign * p0.E() * p3mod) / (2 * p3mod2);
789:
790:   return M4Vec(alpha * p0.Px(), alpha * p0.Py(), alpha * p0.Pz(), E);
791: }
792:
793:
794: // Kinematic transform in process: p1 + p2 -> {p}, where
795: //
796: // p1, p2 are massless spacelike q^2 < 0 => transformed to lightlike q^2 = 0
797: // {p} massive/massless final states with q^2 = m^2
798: //
799: inline void OffShell2LightCone(M4Vec &p1, M4Vec &p2, std::vector<M4Vec> &p) {
800:   const int    N      = p.size();
801:   const int    MAXITER = 15;
802:   const double STOPEPS = 1e-10;
803:
804:   // 4-momentum sum
805:   auto psumfunc = [&]() {
806:     M4Vec sum(0, 0, 0, 0);
807:     for (const auto &i : aux::indices(p)) { sum += p[i]; }
808:     return sum;
809:   };
810:
811:   // Energy sum
812:   auto Esumfunc = [&]() {
813:     double sum = 0.0;
814:     for (const auto &i : aux::indices(p)) { sum += p[i].E(); }
815:     return sum;
816:   };
817:
818:   /*
819:     // Fractions
820:     auto Efrac = [&] () {
821:       const double Esum = Esumfunc();
822:       std::vector<double> f(p.size());
823:       for (const auto& i : aux::indices(p)) { f[i] = p[i].E() / Esum; }
824:       return f;
825:     };
826:   */
827:
828:   // ------------------------------------------------------------------
829:   // Set initial state spacelike (q^2 < 0) particles to lightcone by
830:   // conserving 3-momentum and thus increasing energy
```



```
831:   p1.SetE(p1.P3mod());
832:   p2.SetE(p2.P3mod());
833:   // ------------------------------------------------------------------
834:
835:   // Get sum
836:   const M4Vec q = p1 + p2;
837:
838:   // Normalize \vec{q} to unit length
839:   const std::vector<double> qvec = gra::matoper::Unit(q.P3());
840:
841:   // Final state masses
842:   std::vector<double> m(p.size());
843:   for (const auto &i : aux::indices(p)) { m[i] = p[i].M(); }
844:
845:   int iter = 0;
846:
847:   // for (const auto& i : aux::indices(p)) { p[i].Print(); }
848:   // std::cout << "iter =" << iter << "  " << (q - psumfunc()).M2() << std::endl;
849:
850:   while (true) {
851:     // ----------------------------------------------------
852:     // 3-Momentum scaling and energy subtraction step
853:
854:     // New energy difference
855:     const double dE = Esumfunc() - q.E();
856:
857:     // Current energy fractions
858:     // std::vector<double> f = Efrac();
859:
860:     for (const auto &i : aux::indices(p)) {
861:       const double E = p[i].E() - dE / N;  // Scaling distributed by simple 1/N
862:       // const double E = p[i].E() - dE * f[i]; // Scaling distributed by fractions
863:
864:       const double a = gra::math::msqrt(E * E - m[i] * m[i]) / p[i].P3mod();
865:       p[i].Set(a * p[i].Px(), a * p[i].Py(), a * p[i].Pz(), E);
866:     }
867:
868:     // ----------------------------------------------------
869:     // 3-Momentum subtraction step
870:
871:     // New 3-momentum difference
872:     const std::vector<double> D3 = (psumfunc() - q).P3();
873:
874:     // Current energy fractions
875:     // f = Efrac();
876:
877:     for (const auto &i : aux::indices(p)) {
878:       std::vector<double> D3_this = D3;
879:       gra::matoper::ScaleVector(D3_this, 1.0 / N);  // Scaling distributed by simple 1/N
880:       // gra::matoper::ScaleVector(D3_this, f[i]); // Scaling distributed by fractions
881:
882:       p[i].SetP3(gra::matoper::Minus(p[i].P3(), D3_this));
883:       p[i].SetE(gra::math::msqrt(p[i].P3mod2() + m[i] * m[i]));
884:     }
885:
886:     // ----------------------------------------------------
887:     // 3-Momentum rotation step
888:     /*
889:     // Add and normalize to unit length, find rotation
890:     const std::vector<double> avec = gra::matoper::Unit(psumfunc().P3());
891:     const gra::MMatrix<double> R = gra::kinematics::RotOnetoOne(avec, qvec);
892:
893:     // Rotate
894:     for (const auto& i : aux::indices(p)) { p[i].SetP3( R * p[i].P3()); }
895:     */
896:     // ----------------------------------------------------
897:
898:     ++iter;
899:     const double IVM = std::abs((q - (psumfunc())).M2());
900:     if (iter > MAXITER || IVM < STOPEPS) { break; }
901:
902:     // std::cout << "iter =" << iter << "  " << IVM << std::endl;
903:   }
904:
905:   // for (const auto& i : aux::indices(p)) { p[i].Print(); }
906:   // std::cout << " ---------------------------------------------------- " << std::endl;
907: }
908:
909:
910: // A unified Lorentz Transform function to 'frametype' give below:
911: //
912: // "CS" : Collins-Soper
913: // "AH" : Anti-Helicity (Anti-CS)
```



```
914: // "HE" : Helicity
915: // "PG" : Pseudo-Gottfried-Jackson
916: //
917: // Collins-Soper: Quantization z-axis defined by the bi-sector vector between
918: // initial state p1 and (-p2) (NEGATIVE) directions in the (resonance) system rest frame,
919: // where p1 and p2 are the initial state proton 3-momentum.
920: //
921: // Anti-Helicity: Quantization z-axis defined by the bisector vector between
922: // initial state p1 and (p2) (POSITIVE) directions in the (resonance) system rest frame,
923: // where p1 and p2 are the initial state proton 3-momentum.
924: //
925: // Helicity: Quantization axis defined by the resonance system
926: // 3-momentum vector in the colliding beams frame (lab frame).
927: //
928: // Pseudo-Gottfried-Jackson: Quantization axis defined by the initial state
929: // proton p1 (or p2) 3-momentum vector in the (resonance) system rest frame.
930: //
931:
932: template <typename T>
933: inline void LorentFramePrepare(const std::vector<T> &p, const T &X, const T &pbeam1,
934:                                const T &pbeam2, T &pb1boost, T &pb2boost, std::vector<T> &pboost) {
935:   const double M = X.M();
936:
937:   // 2. Boost each particle to the system rest frame
938:   pboost = p;
939:   for (const auto &k : gra::aux::indices(p)) {
940:     gra::kinematics::LorentzBoost(X, M, pboost[k], -1);  // note minus sign
941:   }
942:
943:   // 3. Boost initial state protons
944:   pb1boost = pbeam1;
945:   pb2boost = pbeam2;
946:   gra::kinematics::LorentzBoost(X, M, pb1boost, -1);  // note minus sign
947:   gra::kinematics::LorentzBoost(X, M, pb2boost, -1);  // note minus sign
948: }
949:
950:
951: template <typename T>
952: inline void LorentzFrame(std::vector<T> &pfout, const T &pb1boost, const T &pb2boost,
953:                          const std::vector<T> &pfboost, const std::string &frametype,
954:                          int direction) {
955:   const std::vector<double> pb1boost3 = pb1boost.P3();
956:   const std::vector<double> pb2boost3 = pb2boost.P3();
957:
958:   // Frame rotation x-y-z-axes
959:   std::vector<double> zaxis;
960:   std::vector<double> yaxis;
961:   std::vector<double> xaxis;
962:
963:   // @@ NON-ROTATED FRAME AXIS DEFINITION @@
964:   if (frametype == "CM") {
965:     zaxis = {0, 0, 1};
966:     yaxis = {0, 1, 0};
967:   }
968:   // @@ COLLINS-SOPER FRAME POLARIZATION AXIS DEFINITION @@
969:   else if (frametype == "CS" || frametype == "G1") {
970:     zaxis = gra::matoper::Unit(
971:         gra::matoper::Minus(gra::matoper::Unit(pb1boost3), gra::matoper::Unit(pb2boost3)));
972:   }
973:   // @@ ANTI-HELICITY FRAME POLARIZATION AXIS DEFINITION @@
974:   else if (frametype == "AH" || frametype == "G2") {
975:     zaxis = gra::matoper::Unit(
976:         gra::matoper::Plus(gra::matoper::Unit(pb1boost3), gra::matoper::Unit(pb2boost3)));
977:   }
978:   // @@ HELICITY FRAME POLARIZATION AXIS DEFINITION @@
979:   else if (frametype == "HE" || frametype == "G3") {
980:     zaxis = gra::matoper::Unit(gra::matoper::Negat(gra::matoper::Plus(pb1boost3, pb2boost3)));
981:   }
982:   // @@ PSEUDO-GOTTFRIED-JACKSON AXIS DEFINITION: [1] or [2] @@
983:   else if (frametype == "PG" || frametype == "G4") {
984:     if (direction == -1) {
985:       zaxis = gra::matoper::Unit(pb1boost3);
986:     } else if (direction == 1) {
987:       zaxis = gra::matoper::Unit(pb2boost3);
988:     } else {
989:       throw std::invalid_argument("gra::kinematics::LorentzFrame: Invalid direction <" +
990:                                   std::to_string(direction) + ">");
991:     }
992:   } else {
993:     throw std::invalid_argument("gra::kinematics::LorentzFrame: Unknown frame <" + frametype + ">");
994:   }
995:
996:   // y-axis
```



```
997:   if (frametype != "CM") {
998:     yaxis = gra::matoper::Unit(
999:         gra::matoper::Cross(gra::matoper::Unit(pb1boost3), gra::matoper::Unit(pb2boost3)));
1000:   }
1001:
1002:   // x-axis
1003:   xaxis = gra::matoper::Unit(gra::matoper::Cross(yaxis, zaxis));  // x = y [cross product] z
1004:
1005:   // Create SO(3) rotation matrix for the new coordinate axes
1006:   const MMatrix<double> R = {xaxis, yaxis, zaxis};  // Axes as rows
1007:
1008:   // Rotate all vectors
1009:   pfout = pfboost;
1010:   for (const auto &k : gra::aux::indices(pfout)) {
1011:     const std::vector<double> p3new = R * pfout[k].P3();        // Spatial part rotation Rp -> p'
1012:     pfout[k] = T(p3new[0], p3new[1], p3new[2], pfout[k].E());  // Full 4-momentum [px; py; pz; E]
1013:   }
1014: }
1015:
1016: // From lab to Anti-Helicity frame
1017: // Quantization z-axis is defined as the bisector of two beams, in the rest
1018: // frame of the resonance
1019: //
1020: // Input:     p  =  Set of 4-momentum to be transformed
1021: //            X  =  System 4-momentum
1022: //            p1 =  Beam 1 4-momentum
1023: //            p2 =  Beam 2 4-momentum
1024: //
1025: template <typename T>
1026: inline void AHframe(std::vector<T> &p, const T &X, const T &p1, const T &p2, bool DEBUG = false) {
1027:   // ****************************************************************
1028:   if (DEBUG) {
1029:     printf("\n\n ::ANTI-HELICITY FRAME:: \n");
1030:     printf("AHframe:: Daughters in LAB FRAME: \n");
1031:     for (const auto &i : gra::aux::indices(p)) { p[i].Print(); }
1032:   }
1033:   // ****************************************************************
1034:
1035:   // Boost particles to the central system rest frame
1036:   for (const auto &i : gra::aux::indices(p)) {
1037:     gra::kinematics::LorentzBoost(X, X.M(), p[i], -1);  // Note the minus sign
1038:   }
1039:
1040:   T p1b = p1;
1041:   T p2b = p2;
1042:
1043:   // Boost the beam particles
1044:   gra::kinematics::LorentzBoost(X, X.M(), p1b, -1);  // Note the minus sign
1045:   gra::kinematics::LorentzBoost(X, X.M(), p2b, -1);  // Note the minus sign
1046:
1047:   // Now get the 3-momentum
1048:   const std::vector<double> pb1boost3 = p1b.P3();
1049:   const std::vector<double> pb2boost3 = p2b.P3();
1050:
1051:   // Frame rotation x-y-z-axes
1052:   std::vector<double> zaxis;
1053:   std::vector<double> yaxis;
1054:   std::vector<double> xaxis;
1055:
1056:   // Anti-Helicity positive bisector vector
1057:   zaxis = gra::matoper::Unit(
1058:       gra::matoper::Plus(gra::matoper::Unit(pb1boost3), gra::matoper::Unit(pb2boost3)));
1059:
1060:   yaxis = gra::matoper::Unit(
1061:       gra::matoper::Cross(gra::matoper::Unit(pb1boost3), gra::matoper::Unit(pb2boost3)));
1062:
1063:   xaxis = gra::matoper::Unit(gra::matoper::Cross(yaxis, zaxis));  // x = y [cross product] z
1064:
1065:   // Create SO(3) rotation matrix for the new coordinate axes
1066:   const MMatrix<double> R = {xaxis, yaxis, zaxis};  // Axes as rows
1067:
1068:   // Rotate all vectors
1069:   for (const auto &k : gra::aux::indices(p)) {
1070:     const std::vector<double> p3new = R * p[k].P3();   // Spatial part rotation Rp -> p'
1071:     p[k] = T(p3new[0], p3new[1], p3new[2], p[k].E());  // Full 4-momentum [px; py; pz; E]
1072:   }
1073:
1074:   // ****************************************************************
1075:   if (DEBUG) {
1076:     printf("AHframe:: Daughters in AH FRAME: \n");
1077:     for (const auto &i : gra::aux::indices(p)) { p[i].Print(); }
1078:   }
1079:   // ****************************************************************
```

```
}

// From lab to Collins-Soper frame
// Quantization z-axis is defined as the bisector of two beams, in the rest
// frame of the resonance
//
// Input:     p  =  Set of 4-momentum to be transformed
//            X  =  System 4-momentum
//            p1 =  Beam 1 4-momentum
//            p2 =  Beam 2 4-momentum
//
template <typename T>
inline void CSframe(std::vector<T> &p, const T &X, const T &p1, const T &p2, bool DEBUG = false) {
  // ************************************************************
  if (DEBUG) {
    printf("\n\n ::COLLINS-SOPER FRAME:: \n");
    printf("CSframe:: Daughters in LAB FRAME: \n");
    for (const auto &i : gra::aux::indices(p)) { p[i].Print(); }
  }
  // ************************************************************

  // Boost particles to the central system rest frame
  for (const auto &i : gra::aux::indices(p)) {
    gra::kinematics::LorentzBoost(X, X.M(), p[i], -1);  // Note the minus sign
  }

  T p1b = p1;
  T p2b = p2;

  // Boost the beam particles
  gra::kinematics::LorentzBoost(X, X.M(), p1b, -1);  // Note the minus sign
  gra::kinematics::LorentzBoost(X, X.M(), p2b, -1);  // Note the minus sign

  // Now get the 3-momentum
  const std::vector<double> p1b3 = p1b.P3();
  const std::vector<double> p2b3 = p2b.P3();

  // Frame rotation x-y-z-axes
  std::vector<double> zaxis;
  std::vector<double> yaxis;
  std::vector<double> xaxis;

  // Collins-Soper bisector vector
  zaxis = gra::matoper::Unit(
      gra::matoper::Minus(gra::matoper::Unit(p1b3), gra::matoper::Unit(p2b3)));

  /* ALTERNATIVE WAY, but gives random reflection (rotation around Z by PI)

  M4Vec bijector;
  bijector.SetP3(zaxis);

  // Now get the rotation angle, note the minus
  const double Z_angle = -bijector.Phi();
  const double Y_angle = -bijector.Theta();

  // Rotate final states
  for (const auto &i : gra::aux::indices(p)) {
    gra::kinematics::RotateZ(p[i], Z_angle);
    gra::kinematics::RotateY(p[i], Y_angle);
    gra::kinematics::RotateZ(p[i], math::PI);
  }
  */

  yaxis = gra::matoper::Unit(
      gra::matoper::Cross(gra::matoper::Unit(p1b3), gra::matoper::Unit(p2b3)));

  xaxis = gra::matoper::Unit(gra::matoper::Cross(yaxis, zaxis));  // x = y [cross product] z

  // Create SO(3) rotation matrix for the new coordinate axes
  const M4Matrix<double> R = {xaxis, yaxis, zaxis};  // Axes as rows

  // Rotate all vectors
  for (const auto &k : gra::aux::indices(p)) {
    const std::vector<double> p3new = R * p[k].P3();   // Spatial part rotation Rp -> p'
    p[k] = T(p3new[0], p3new[1], p3new[2], p[k].E());  // Full 4-momentum (px, py, pz, E)
  }

  // ************************************************************
  if (DEBUG) {
    printf("CSframe:: Daughters in CS FRAME: \n");
    for (const auto &i : gra::aux::indices(p)) { p[i].Print(); }
  }
```



```
  // ************************************************************
}

// From lab to Gottfried-Jackson frame
// Quantization z-axis spanned by the propagator (momentum transfer vector)
// momentum in the rest frame of the central system
//
// Input:  p         = Set of 4-momentum to be transformed
//         X         = System 4-momentum
//         direction = -1 or 1
//         q1        = 4-momentum transfer vector 1
//         q2        = 4-momentum transfer vector 2
//
template <typename T>
inline void GJframe(std::vector<T> &p, const T &X, int direction, const T &q1, const T &q2,
                    bool DEBUG = false) {
  // ************************************************************
  if (DEBUG) {
    printf("\n\n ::GOTTFRIED-JACKSON FRAME:: \n");

    printf("GJframe:: Daughters in LAB FRAME: \n");
    for (const auto &i : gra::aux::indices(p)) { p[i].Print(); }
  }
  // ************************************************************

  // Boost particles to the central system rest frame
  for (const auto &i : gra::aux::indices(p)) {
    gra::kinematics::LorentzBoost(X, X.M(), p[i], -1);  // Note the minus sign
  }

  // Boost the propagators
  T q1boost = q1;
  T q2boost = q2;
  gra::kinematics::LorentzBoost(X, X.M(), q1boost, -1);  // Note the minus sign
  gra::kinematics::LorentzBoost(X, X.M(), q2boost, -1);  // Note the minus sign

  // Now get the rotation angle, note the minus
  double Z_angle = 0;
  double Y_angle = 0;
  if (direction == -1) {
    Z_angle = -q1boost.Phi();
    Y_angle = -q1boost.Theta();
  } else if (direction == 1) {
    Z_angle = -q2boost.Phi();
    Y_angle = -q2boost.Theta();
  } else {
    throw std::invalid_argument("GJframe: direction not -1 or 1");
  }

  const auto rotate = [&](T &p) {
    gra::kinematics::RotateZ(p, Z_angle);
    gra::kinematics::RotateY(p, Y_angle);
    // gra::kinematics::RotateZ(p, math::PI); // Reflection
  };

  // Rotate final states
  for (const auto &i : gra::aux::indices(p)) { rotate(p[i]); }

  // ************************************************************
  if (DEBUG) {
    printf("GJframe:: Daughters in Gottfried-Jackson FRAME: \n");
    for (const auto &i : gra::aux::indices(p)) { p[i].Print(); }

    // Rotate propagator
    rotate(q1boost);
    rotate(q2boost);

    printf("GJframe:: Propagators are along z-axis \n");
    printf("GJframe:: Propagator 1 in Gottfried-Jackson FRAME: \n");
    q1boost.Print();

    printf("GJframe:: Propagator 2 in Gottfried-Jackson FRAME: \n");
    q2boost.Print();
  }
  // ************************************************************
}


// From lab to Pseudo-Gottfried-Jackson frame
// Quantization z-axis spanned by the beam proton +z (or -z) momentum
// in the rest frame of the central system
//
```



```
// Input:  p             = Set of 4-momentum to be transformed
//         X             = System 4-momentum
//         direction     = -1 or 1
//         p_beam_plus   = 4-momentum of beam1
//         p_beam_minus  = 4-momentum of beam2
//
template <typename T>
inline void PGframe(std::vector<T> &p, const T &X, const int direction, const T &p_beam_plus,
                    const T &p_beam_minus, bool DEBUG = false) {
  // ************************************************************
  if (DEBUG) {
    printf("\n\n ::PSEUDO-GOTTFRIED-JACKSON FRAME:: \n");

    printf("PGframe:: Daughters in LAB FRAME: \n");
    for (const auto &i : gra::aux::indices(p)) { p[i].Print(); }
  }
  // ************************************************************

  // Boost particles to the central system rest frame
  for (const auto &i : gra::aux::indices(p)) {
    gra::kinematics::LorentzBoost(X, X.M(), p[i], -1);  // Note the minus sign
  }

  // Boost initial state protons to the central system rest frame
  T proton_p = p_beam_plus;
  T proton_m = p_beam_minus;

  gra::kinematics::LorentzBoost(X, X.M(), proton_p, -1);  // Note the minus sign
  gra::kinematics::LorentzBoost(X, X.M(), proton_m, -1);  // Note the minus sign

  // Now get the rotation angles, note the minus
  double Z_angle = 0;
  double Y_angle = 0;

  if (direction == -1) {
    Z_angle = -proton_p.Phi();
    Y_angle = -proton_p.Theta();
  } else if (direction == 1) {
    Z_angle = -proton_m.Phi();
    Y_angle = -proton_m.Theta();
  } else {
    printf("PGframe:: Input direction %d not valid (1,-1)!", direction);
    return;
  }

  // Rotation function
  const auto rotate = [&](T &p) {
    gra::kinematics::RotateZ(p, Z_angle);
    gra::kinematics::RotateY(p, Y_angle);
    // gra::kinematics::RotateZ(p, math::PI); // Reflection
  };

  // Rotate final states
  for (const auto &i : gra::aux::indices(p)) { rotate(p[i]); }

  // ************************************************************
  if (DEBUG) {
    printf("PGframe:: Daughters in Pseudo-Gottfried-Jackson FRAME: \n");
    for (const auto &i : gra::aux::indices(p)) { p[i].Print(); }

    // Rotate proton
    rotate(proton_p);

    printf("Protons are on (zx)-plane: \n");
    printf("USER chosen direction along %d beam direction \n", direction);
    printf("  +z Proton in Pseudo-Gottfried-Jackson FRAME: \n");
    proton_p.Print();

    // Rotate other proton
    rotate(proton_m);

    printf("  -z Proton in Pseudo-Gottfried-Jackson FRAME: \n");
    proton_m.Print();

    // Mandelstam invariant
    printf("Mandelstam s = %0.5f \n", (proton_p + proton_m).M());
  }
  // ************************************************************
}

// From lab to the Helicity frame
// Quantization z-axis as the direction of the resonance in the lab frame
```



```
//
// Input:  p = Set of 4-momentum to be transformed
//         X = System 4-momentum
//
template <typename T>
inline void HXframe(std::vector<T> &p, const T &X, bool DEBUG = false) {
  // ************************************************************
  if (DEBUG) {
    printf("\n\n ::HELICITY FRAME:: \n");
    printf("HXframe:: Daughters in LAB FRAME: \n");
    for (const auto &i : gra::aux::indices(p)) { p[i].Print(); }
  }
  // ************************************************************

  // Rotation angles, note the minus
  const double Z_angle = -X.Phi();
  const double Y_angle = -X.Theta();

  const auto rotate = [&](T &p) {
    gra::kinematics::RotateZ(p, Z_angle);
    gra::kinematics::RotateY(p, Y_angle);
    gra::kinematics::RotateZ(p, math::PI);  // Reflection
  };

  for (const auto &i : gra::aux::indices(p)) { rotate(p[i]); }

  // ************************************************************
  if (DEBUG) {
    printf("HXframe:: Daughters in ROTATED LAB FRAME: \n");
    for (const auto &i : gra::aux::indices(p)) { p[i].Print(); }

    T ex(1, 0, 0, 0);
    T ey(0, 1, 0, 0);
    T ez(0, 0, 1, 0);

    // x -> x', y -> y', z -> z'
    rotate(ex);
    rotate(ey);
    rotate(ez);

    printf("HXframe:: AXIS vectors after rotation: \n");

    ex.Print();
    ey.Print();
    ez.Print();
  }
  // ************************************************************

  // Construct the central system 4-momentum in ROTATED FRAME
  T XNEW = X;
  rotate(XNEW);

  // Boost particles to the central system rest frame
  // -> Helicity frame obtained
  for (const auto &i : gra::aux::indices(p)) {
    gra::kinematics::LorentzBoost(XNEW, XNEW.M(), p[i], -1);  // Note the minus sign
  }

  // ************************************************************
  if (DEBUG) {
    printf("HXframe:: Daughters after boost in HELICITY FRAME: \n");
    for (const auto &i : gra::aux::indices(p)) { p[i].Print(); }

    printf("HXframe:: Central system in LAB FRAME: \n");
    X.Print();

    printf("HXframe:: Central system in ROTATED LAB FRAME: \n");
    XNEW.Print();

    printf("\n");
  }
  // ************************************************************
}


// "Rest frame" (no boost, beam axis as z-axis / spin quantization axis)
//
// Input:  p = Set of 4-momentum to be transformed
//         X = System 4-momentum
//
template <typename T>
inline void CMframe(std::vector<T> &p, const T &X, bool DEBUG = false) {
  // ************************************************************
```

./include/Graniitti/MKinematics.h 18/20

```
1412:   if (DEBUG) {
1413:     printf("\n\n ::REST FRAME:: \n");
1414:
1415:     printf("CMframe:: Daughters in LAB FRAME: \n");
1416:     for (const auto &i : gra::aux::indices(p)) { p[i].Print(); }
1417:   }
1418:   // ****************************************************************
1419:
1420:   // Boost particles to the central system rest frame
1421:   for (const auto &i : gra::aux::indices(p)) {
1422:     gra::kinematics::LorentzBoost(X, X.M(), p[i], -1);  // Note the minus sign
1423:   }
1424:
1425:   // ****************************************************************
1426:   if (DEBUG) {
1427:     printf("CMframe:: Daughters in REST FRAME: \n");
1428:     for (const auto &i : gra::aux::indices(p)) { p[i].Print(); }
1429:   }
1430:   // ****************************************************************
1431: }
1432:
1433: }  // namespace kinematics
1434:
1435:
1436: // Lorentz scalars and other common kinematic variables
1437: class LORENTZSCALAR {
1438:  public:
1439:   LORENTZSCALAR() {}
1440:   ~LORENTZSCALAR() {
1441:     delete GlobalSudakovPtr;
1442:     delete GlobalPdfPtr;
1443:   }
1444:
1445:   // ------------------------------------------------------------------
1446:   // Particle Database
1447:   MPDG PDG;
1448:
1449:   // ------------------------------------------------------------------
1450:   // PDF access
1451:   // Sudakov/pdf routines
1452:   MSudakov *GlobalSudakovPtr = nullptr;
1453:
1454:   // Normal pdfs
1455:   std::string  LHAPDFSET    = "null";
1456:   LHAPDF::PDF *GlobalPdfPtr = nullptr;
1457:   int          pdf_trials   = 0;
1458:   // ------------------------------------------------------------------
1459:
1460:   // Cascade sampling forced control initiated by corresponding amplitude functions
1461:   bool   FORCE_FLATMASS2 = false;
1462:   double FORCE_OFFSHELL  = -1;
1463:
1464:   // ------------------------------------------------------------------
1465:
1466:   // Resonances read from JSON input
1467:   std::map<std::string, PARAM_RES> RESONANCES;
1468:
1469:   // Helicity amplitudes returned by amplitude functions,
1470:   // used in screening loop etc.
1471:   std::vector<std::complex<double>> hamp;
1472:
1473:   // Central system decaytree
1474:   std::vector<MDecayBranch> decaytree;
1475:
1476:   // Forward system decaytree
1477:   MDecayBranch decayforward1;
1478:   MDecayBranch decayforward2;
1479:
1480:   // Integral weight container (central system phase space)
1481:   gra::kinematics::MCW DW;
1482:
1483:   // Active in the factorized phase space product (by default, true)
1484:   // This is controlled by the spesific amplitudes
1485:   bool PS_active = true;
1486:
1487:   // Sum containers
1488:   gra::kinematics::MCWSUM DW_sum;
1489:   gra::kinematics::MCWSUM DW_sum_exact;
1490:
1491:   // Initial states
1492:   gra::MParticle beam1;
1493:   gra::MParticle beam2;
1494:
```

./include/Graniitti/MKinematics.h 19/20

```
1495:   // Four momenta of initial and final states
1496:   M4Vec              pbeam1;
1497:   M4Vec              pbeam2;
1498:   std::vector<M4Vec> pfinal;
1499:   std::vector<M4Vec> pfinal_orig;
1500:
1501:   // Basic Lorentz scalars
1502:   double s      = 0.0;
1503:   double t      = 0.0;
1504:   double u      = 0.0;
1505:   double sqrt_s = 0.0;
1506:
1507:   // Sub-energies^2
1508:   double s1 = 0.0;
1509:   double s2 = 0.0;
1510:
1511:   // 4-momentum transfer squared
1512:   double t1 = 0.0;
1513:   double t2 = 0.0;
1514:
1515:   // Sub-Mandelstam variables
1516:   double s_hat = 0.0;
1517:   double t_hat = 0.0;
1518:   double u_hat = 0.0;
1519:
1520:   // Central system
1521:   double m2 = 0.0;
1522:   double Y  = 0.0;
1523:   double Pt = 0.0;
1524:
1525:   // Maximum 10 particles in the final state
1526:   // (2 protons + 8 direct central system)
1527:   double ss[10][10]    = {{0.0}};
1528:   double tt_1[10]      = {0.0};
1529:   double tt_2[10]      = {0.0};
1530:   double tt_xy[10][10] = {{0.0}};
1531:
1532:   // Longitudinal momentum losses
1533:   double x1 = 0.0;
1534:   double x2 = 0.0;
1535:
1536:   // Bjorken-x
1537:   double xbj1 = 0.0;
1538:   double xbj2 = 0.0;
1539:
1540:   // Forward proton excitation tag
1541:   bool excite1 = false;
1542:   bool excite2 = false;
1543:
1544:   // Propagators from proton1 (up) and proton2 (down)
1545:   M4Vec q1;
1546:   M4Vec q2;
1547:
1548:   // Propagator pt
1549:   double qt1 = 0.0;
1550:   double qt2 = 0.0;
1551: };
1552:
1553: // Generator cut (default parameters set here)
1554: struct GENCUT {
1555:   // Continuum phase space <C>
1556:   double rap_min = -9.0;
1557:   double rap_max = 9.0;
1558:   double kt_min  = 0.0;
1559:   double kt_max  = -1.0;  // Keep at -1 for user setup trigger
1560:
1561:   // Factorized phase space <F>
1562:   double Y_min = -9.0;
1563:   double Y_max = 9.0;
1564:   double M_min = 0.0;
1565:   double M_max = 0.0;
1566:
1567:   // Both <C> and <F> class forward legs
1568:   double forward_pt_min = -1.0;  // Keep at -1 for user setup trigger
1569:   double forward_pt_max = -1.0;  // Keep at -1 for user setup trigger
1570:
1571:   // ---------------------------------------
1572:
1573:   // Quasi-Elastic phase space <Q> or forward excitation
1574:   double XI_min = 0.0;
1575:   double XI_max = 1.0;
1576: };
1577:
```

./include/Graniitti/MKinematics.h 20/20

```
1578: // Fiducial cuts (default parameters set here)
1579: struct FIDCUT {
1580:   bool active = false;
1581:
1582:   // "Central particles"
1583:   double eta_min = -30.0;
1584:   double eta_max = 30.0;
1585:
1586:   double rap_min = -30.0;
1587:   double rap_max = 30.0;
1588:
1589:   double pt_min = 0.0;
1590:   double pt_max = 1000000.0;
1591:
1592:   double Et_min = 0.0;
1593:   double Et_max = 1000000.0;
1594:
1595:   // "Central system"
1596:   double M_min = 0.0;
1597:   double M_max = 1000000.0;
1598:
1599:   double Y_min = -30.0;
1600:   double Y_max = 30.0;
1601:
1602:   double Pt_min = 0.0;
1603:   double Pt_max = 1000000.0;
1604:
1605:   // "Forward system"
1606:   double forward_t_min = 0.0;
1607:   double forward_t_max = 1000000.0;
1608:
1609:   double forward_M_min = 0.0;
1610:   double forward_M_max = 1000000.0;
1611: };
1612:
1613: // For veto
1614: struct VETODOMAIN {
1615:   double eta_min = -30.0;
1616:   double eta_max = 30.0;
1617:
1618:   double pt_min = 0.0;
1619:   double pt_max = 1000000.0;
1620: };
1621:
1622: struct VETOCUT {
1623:   bool                      active = false;
1624:   std::vector<VETODOMAIN> cuts;
1625: };
1626:
1627:
1628: }  // namespace gra
1629:
1630: #endif
```

./include/Graniitti/MFragment.h 1/1

```
   1: // Toy fragmentation class
   2: //
   3: // (c) 2017-2020 Mikael Mieskolainen
   4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
   5:
   6: #ifndef MFRAGMENTATION_H
   7: #define MFRAGMENTATION_H
   8:
   9: // C++
  10: #include <complex>
  11: #include <map>
  12: #include <random>
  13: #include <vector>
  14:
  15: // Own
  16: #include "Graniitti/MPDG.h"
  17: #include "Graniitti/MRandom.h"
  18:
  19: namespace gra {
  20: class MFragment {
  21:  public:
  22:   MFragment() {}
  23:   ~MFragment() {}
  24:
  25:   static double TubeFragment(const M4Vec &mother, double M0, const std::vector<double> &m,
  26:                              std::vector<M4Vec> &p, double q, double T, double lambda, double maxpt,
  27:                              MRandom &rng, const std::string &pt_distribution);
  28:
  29:   static bool SolveAlpha(double &alpha, double M0, const std::vector<double> &m,
  30:                          const std::valarray<double> &mt, const std::valarray<double> &y);
  31:
  32:   static void ExpPowRND(double q, double T, double maxpt, const std::vector<double> &mass,
  33:                         std::vector<double> &x, MRandom &rng);
  34:
  35:   static void GetDecayStatus(const std::vector<int> &pdgcode, std::vector<bool> &isstable);
  36:
  37:   static void GetForwardMass(double &mass1, double &mass2, bool &excite1, bool &excite2,
  38:                              unsigned int excite, MRandom &random);
  39:   static void GetSingleForwardMass(double &mass, MRandom &random);
  40:   static void NstarDecayTable(int Q, double m0, std::vector<int> &pdgcode, MRandom &rng);
  41:
  42:   static bool PickParticles(double M, unsigned int N, int B, int S, int Q,
  43:                             std::vector<double> &mass, std::vector<int> &pdgcode, const MPDG &PDG,
  44:                             MRandom &rng);
  45: };
  46:
  47: }  // namespace gra
  48:
  49: #endif
```

./include/Graniitti/MDimArray.h 1/1

```
 1: // Multidimensional fixed size arrays via template alias technique
 2: //
 3: //
 4: // Usage:
 5: // MultiDimArray<int, 3, 2> arr {1, 2, 3, 4, 5, 6};
 6: // assert(arr[1][1] == 4);
 7: //
 8: // (c) 2017-2020 Mikael Mieskolainen
 9: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
10:
11: #ifndef MDIMARRAY_H
12: #define MDIMARRAY_H
13:
14: #include <array>
15: #include <complex>
16:
17: namespace gra {
18: // Note the index order below, in order to get the row major format
19: //
20: template <typename T, size_t D1, size_t D2, size_t... DN>
21: struct GetArray {
22:   using type = std::array<typename GetArray<T, D2, DN...>::type, D1>;
23: };
24:
25: template <typename T, size_t D1, size_t D2>
26: struct GetArray<T, D1, D2> {
27:   using type = std::array<std::array<T, D2>, D1>;
28: };
29:
30: template <typename T, size_t D1, size_t D2, size_t... DN>
31: using MDimArray = typename GetArray<T, D1, D2, DN...>::type;
32:
33: }  // namespace gra
34:
35: #endif
```

./include/Graniitti/MSpin.h 1/1

```
 1: // Spin polarization and correlation functions
 2: //
 3: // (c) 2017-2020 Mikael Mieskolainen
 4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
 5:
 6: #ifndef MSPIN_H
 7: #define MSPIN_H
 8:
 9: // C++
10: #include <complex>
11: #include <random>
12: #include <vector>
13:
14: // Own
15: #include "Graniitti/MKinematics.h"
16: #include "Graniitti/MRandom.h"
17:
18: namespace gra {
19: namespace spin {
20:
21: MMatrix<std::complex<double>> CalculateFMatrix(const MDecayBranch &branch);
22: void                           TreeRecursion(MDecayBranch &branch);
23: void TensorTree(const MDecayBranch &branch, MMatrix<std::complex<double>> &out);
24:
25: std::complex<double> ProdAmp(const gra::LORENTZSCALAR &lts, gra::PARAM_RES &res);
26: std::complex<double> DecayAmp(gra::LORENTZSCALAR &lts, gra::PARAM_RES &res);
27: void GetRhoRotation(const gra::LORENTZSCALAR &lts, const std::string &FRAME, double &theta_R,
28:                      double &phi_R);
29:
30: MMatrix<std::complex<double>> DMatrix(double J, double theta_mother, double phi_mother);
31: MMatrix<std::complex<double>> fMatrix(const MMatrix<std::complex<double>> &T, double J, double s1,
32:                                       double s2, double theta, double phi);
33:
34: std::vector<double> SpinProjections(double J);
35: void InitTMatrix(gra::HELMatrix &hc, const gra::MParticle &p, const gra::MParticle &p1,
36:                   const gra::MParticle &p2);
37:
38: // Spin-Statistics
39: bool BoseSymmetry(int l, int s);
40: bool FermiSymmetry(int l, int s);
41:
42: // Spin algebra functions
43: double               ClebschGordan(double j1, double j2, double m1, double m2, double j, double m);
44: double               CGrules(double j1, double j2, double m1, double m2, double j, double m);
45: bool                 chalfint(double x);
46: bool                 cint(double x);
47: bool                 isequal(double x, double y);
48: double               W3j(double j1, double j2, double j3, double m1, double m2, double m3);
49: bool                 W3jrules(double j1, double j2, double j3, double m1, double m2, double m3);
50: double               TriangleCoeff(double j1, double j2, double j3);
51: std::complex<double> WignerD(double theta, double phi, double m, double mp, double J);
52: double               WignerSmall(double theta, double m, double mp, double J);
53:
54: // Density matrix functions
55: bool                           Positivity(const MMatrix<std::complex<double>> &rho, unsigned int J);
56: MMatrix<std::complex<double>> RandomRho(unsigned int J, bool parity, MRandom &rng);
57: double                        VonNeumannEntropy(const MMatrix<std::complex<double>> &rho);
58:
59: }  // namespace spin
60: }  // namespace gra
61:
62: #endif
```

./include/Graniitti/MUserHistograms.h 1/1

```
 1: // Container class for different type of histograms
 2: //
 3: // (c) 2017-2020 Mikael Mieskolainen
 4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
 5:
 6: #ifndef MUSERHISTOGRAMS_H
 7: #define MUSERHISTOGRAMS_H
 8:
 9: // C++
10: #include <complex>
11: #include <vector>
12:
13: // Own
14: #include "Graniitti/MH1.h"
15: #include "Graniitti/MH2.h"
16: #include "Graniitti/MKinematics.h"
17:
18: namespace gra {
19: class MUserHistograms {
20:  public:
21:   // Constructor, destructor
22:   MUserHistograms() {}
23:   ~MUserHistograms() {}
24:
25:   void InitHistograms();
26:   void FillHistograms(double totalweight, const gra::LORENTZSCALAR &scalar);
27:   void PrintHistograms();
28:   void SetHistograms(unsigned int in) { HIST = in; }
29:   void FillCosThetaPhi(double totalweight, const gra::LORENTZSCALAR &scalar);
30:
31:   // Histograms indexed by name std::string
32:   std::map<std::string, MH1<double>> h1;
33:   std::map<std::string, MH2>         h2;
34:
35:   unsigned int HIST = 0;  // Histogramming level
36: };
37:
38: }  // namespace gra
39:
40: #endif
```

./include/Graniitti/MMatrix.h 1/5

```
 1: // Minimal templated matrix class [HEADER ONLY class]
 2: //
 3: //
 4: // (c) 2017-2020 Mikael Mieskolainen
 5: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
 6:
 7: #ifndef MMATRIX_H
 8: #define MMATRIX_H
 9:
10: #include <initializer_list>
11: #include <iomanip>
12: #include <iostream>
13:
14: namespace gra {
15:
16: // Constant size matrix
17: template <typename T>
18: class MMatrix {
19:  public:
20:   MMatrix() : rows(0), cols(0) { data = nullptr; }
21:
22:   MMatrix(std::size_t r, std::size_t c) {
23:     rows = r;
24:     cols = c;
25:     data = new T[rows * cols];
26:     std::fill(data, data + rows * cols, T());  // No initialization
27:   }
28:   MMatrix(std::size_t r, std::size_t c, T value) {
29:     rows = r;
30:     cols = c;
31:     data = new T[rows * cols];
32:     std::fill(data, data + rows * cols, T(value));  // Initialization
33:   }
34:   MMatrix(std::size_t r, std::size_t c, const std::string &special) {
35:     rows = r;
36:     cols = c;
37:     data = new T[rows * cols];
38:
39:     if (special == "eye") {
40:       std::fill(data, data + rows * cols, T(0.0));
41:       Identity();
42:     } else if (special == "minkowski") {
43:       std::fill(data, data + rows * cols, T(0.0));
44:       Minkowski();
45:     } else {
46:       throw std::invalid_argument("MMatrix: Unknown initialization string:" + special);
47:     }
48:   }
49:   // Set matrix to identity [diagonal = 1, otherwise 0]
50:   void Identity() {
51:     for (std::size_t i = 0; i < rows; ++i) {
52:       for (std::size_t j = 0; j < cols; ++j) { this->operator()(i, j) = (i == j) ? 1.0 : 0.0; }
53:     }
54:   }
55:   // Set matrix to Minkowski metric
56:   void Minkowski() {
57:     for (std::size_t i = 0; i < rows; ++i) {
58:       for (std::size_t j = 0; j < cols; ++j) {
59:         double sign          = (i > 0 && j > 0) ? -1.0 : 1.0;
60:         this-> operator()(i, j) = (i == j) ? sign * 1.0 : 0.0;
61:       }
62:     }
63:   }
64:
65:   // For initializing with a = {row0-vector, row1-vector, ...}
66:   // where each row is std::vector<T>
67:   //
68:   // Row-major order!
69:   MMatrix(std::initializer_list<std::vector<T>> list) {
70:     rows = list.size();
71:     cols = list.begin()->size();
72:     data = new T[rows * cols];
73:
74:     std::size_t i = 0;
75:     for (const auto &v : list) {
76:       for (std::size_t j = 0; j < cols; ++j) { data[cols * i + j] = v[j]; }
77:       ++i;
78:     }
79:   }
80:
81:   // For initializing with a = { {}, {}, {} }
82:   //
83:   // Row-major order!
```

```
84:   MMatrix(std::initializer_list<std::initializer_list<T>> list) {
85:     rows = list.size();
86:     cols = list.begin()->size();
87:     data = new T[rows * cols];
88:
89:     std::size_t i = 0;
90:     for (const auto &v : list) {
91:       std::size_t j = 0;
92:       for (const auto &w : v) {
93:         data[cols * i + j] = w;
94:         ++j;
95:       }
96:       ++i;
97:     }
98:   }
99:
100:  // Copy constructor
101:  MMatrix(const MMatrix &a) {
102:    rows = a.rows;
103:    cols = a.cols;
104:    data = new T[rows * cols];
105:
106:    // Copy all elements
107:    Copy(a);
108:  }
109:
110:  ~MMatrix() {
111:    // delete dynamically allocated memory
112:    delete[] data;
113:  }
114:
115:  // Assignment operator
116:  MMatrix &operator=(const MMatrix &rhs) {
117:    if (data != rhs.data && rhs.data != nullptr) {
118:      ReSize(rhs.rows, rhs.cols);
119:      Copy(rhs);
120:    }
121:    return *this;
122:  }
123:  // For indexing with [i][j]
124:  // Row-major order!
125:  T *      operator[](const std::size_t &row) { return data + cols * row; }
126:  const T *operator[](const std::size_t &row) const { return data + cols * row; }
127:
128:  // Row-major order!
129:  T &operator()(std::size_t i, std::size_t j) {
130:    if (i >= rows || j >= cols) {
131:      throw std::out_of_range("MMatrix:: Index over matrix dimensions!");
132:    }
133:    return data[cols * i + j];
134:  }
135:  const T &operator()(std::size_t i, std::size_t j) const {
136:    if (i >= rows || j >= cols) {
137:      throw std::out_of_range("MMatrix:: Index over matrix dimensions!");
138:    }
139:    return data[cols * i + j];
140:  }
141:
142:  // ------------------------------------------------------------------
143:  // Add to the left
144:  MMatrix &operator+=(const MMatrix &rhs) {
145:    for (std::size_t i = 0; i < rows; ++i) {
146:      for (std::size_t j = 0; j < cols; ++j) { this->operator()(i, j) += rhs[i][j]; }
147:    }
148:    return *this;
149:  }
150:  // Subtract to the left
151:  MMatrix &operator-=(const MMatrix &rhs) {
152:    for (std::size_t i = 0; i < rows; ++i) {
153:      for (std::size_t j = 0; j < cols; ++j) { this->operator()(i, j) -= rhs[i][j]; }
154:    }
155:    return *this;
156:  }
157:
158:  // ------------------------------------------------------------------
159:
160:  // Return negated matrix
161:  MMatrix operator-() const {
162:    MMatrix out(rows, cols);
163:    for (std::size_t i = 0; i < rows; ++i) {
164:      for (std::size_t j = 0; j < cols; ++j) { out[i][j] = -this->operator()(i, j); }
165:    }
166:    return out;
```



```
167:    }
168:
169:  // Add two matrices
170:  MMatrix operator+(const MMatrix &rhs) const {
171:    MMatrix out(rows, cols);
172:    if (rows != rhs.size_row() || cols != rhs.size_col()) {
173:      throw std::invalid_argument("MMatrix:: Matrix + Matrix with invalid dimensions");
174:    }
175:
176:    for (std::size_t i = 0; i < rows; ++i) {
177:      for (std::size_t j = 0; j < cols; ++j) { out[i][j] = this->operator()(i, j) + rhs[i][j]; }
178:    }
179:    return out;
180:  }
181:  // Subtract two matrices
182:  MMatrix operator-(const MMatrix &rhs) const {
183:    MMatrix out(rows, cols);
184:    if (rows != rhs.size_row() || cols != rhs.size_col()) {
185:      throw std::invalid_argument("MMatrix:: Matrix - Matrix with invalid dimensions");
186:    }
187:
188:    for (std::size_t i = 0; i < rows; ++i) {
189:      for (std::size_t j = 0; j < cols; ++j) { out[i][j] = this->operator()(i, j) - rhs[i][j]; }
190:    }
191:    return out;
192:  }
193:  // ------------------------------------------------------------------
194:
195:  // ------------------------------------------------------------------
196:  // We do not allow addition or substraction by scalar, for dimensional
197:  // "safety" reasons (can lead to accidental results), thus only multiplicative
198:  // operations.
199:
200:  // Multiply by a scalar
201:  MMatrix operator*(const T &rhs) const {
202:    MMatrix out(rows, cols);
203:    for (std::size_t i = 0; i < rows; ++i) {
204:      for (std::size_t j = 0; j < cols; ++j) { out[i][j] = this->operator()(i, j) * rhs; }
205:    }
206:    return out;
207:  }
208:
209:  // Divide by a scalar
210:  MMatrix operator/(const T &rhs) const {
211:    MMatrix out(rows, cols);
212:    for (std::size_t i = 0; i < rows; ++i) {
213:      for (std::size_t j = 0; j < cols; ++j) { out[i][j] = this->operator()(i, j) / rhs; }
214:    }
215:    return out;
216:  }
217:  // ------------------------------------------------------------------
218:
219:  // Matrix [this] * Vector [rhs] multiplication
220:  std::vector<T> operator*(const std::vector<T> &rhs) const {
221:    if (rhs.size() != cols) {
222:      throw std::invalid_argument(
223:          "MMatrix:: matrix * vector product with invalid dimension on rhs");
224:    }
225:
226:    std::vector<T> out(rows, 0.0);  // Init!
227:    for (std::size_t i = 0; i < rows; ++i) {
228:      for (std::size_t j = 0; j < cols; ++j) { out[i] += this->operator()(i, j) * rhs[j]; }
229:    }
230:    return out;
231:  }
232:
233:  // Matrix [this] * Matrix [rhs] multiplication
234:  MMatrix operator*(const MMatrix &rhs) const {
235:    const std::size_t n = rows;
236:    const std::size_t m = cols;
237:    const std::size_t p = rhs.size_col();
238:
239:    MMatrix<T> C(n, p, 0.0);  // Note initialization to 0.0!
240:
241:    if (cols != rhs.size_row()) {
242:      throw std::invalid_argument("MMatrix:: matrix * matrix product with invalid dimensions");
243:    }
244:
245:    // Over [i,j] of C
246:    for (std::size_t i = 0; i < n; ++i) {
247:      for (std::size_t j = 0; j < p; ++j) {
248:        for (std::size_t k = 0; k < m; ++k) {
249:          C[i][j] += this->operator()(i, k) * rhs[k][j];  // notice plus
```



```
250:        }
251:      }
252:    }
253:    return C;
254:  }
255:  // ------------------------------------------------------------------
256:
257:  // Sum all elements
258:  T Sum() const {
259:    T sum(0.0);
260:    for (std::size_t i = 0; i < rows; ++i) {
261:      for (std::size_t j = 0; j < cols; ++j) { sum += this->operator()(i, j); }
262:    }
263:    return sum;
264:  }
265:
266:  // Frobenius norm
267:  double FrobNorm() const { return std::sqrt(FrobNorm2()); }
268:
269:  // Frobenius norm squared
270:  double FrobNorm2() const {
271:    double sum = 0.0;
272:    for (std::size_t i = 0; i < rows; ++i) {
273:      for (std::size_t j = 0; j < cols; ++j) {
274:        const double value = std::abs(this->operator()(i, j));
275:        sum += value * value;  // |a_ij|^2
276:      }
277:    }
278:    return sum;
279:  }
280:
281:  // Get transposed matrix
282:  MMatrix Transpose() const {
283:    MMatrix out(cols, rows);
284:    for (std::size_t i = 0; i < rows; ++i) {
285:      for (std::size_t j = 0; j < cols; ++j) { out[j][i] = this->operator()(i, j); }
286:    }
287:    return out;
288:  }
289:  // Get conjugate transposed matrix (dagger)
290:  MMatrix ConjTranspose() const {
291:    MMatrix out(cols, rows);
292:    for (std::size_t i = 0; i < rows; ++i) {
293:      for (std::size_t j = 0; j < cols; ++j) { out[j][i] = std::conj(this->operator()(i, j)); }
294:    }
295:    return out;
296:  }
297:  MMatrix Dagger() const { return ConjTranspose(); }
298:
299:  // Get diagonal vector
300:  std::vector<T> GetDiag() const {
301:    if (rows != cols) {
302:      throw std::invalid_argument("MMatrix::GetDiag: Only defined for square matrices, rows = " +
303:                                  std::to_string(rows) + " , cols = " + std::to_string(cols));
304:    }
305:    std::vector<T> d(rows);
306:    for (std::size_t i = 0; i < rows; ++i) { d[i] = this->operator()(i, i); }
307:    return d;
308:  }
309:
310:  // Get trace
311:  T Trace() const {
312:    if (rows != cols) {
313:      throw std::invalid_argument("MMatrix::Trace: Only defined for square matrices, rows = " +
314:                                  std::to_string(rows) + " , cols = " + std::to_string(cols));
315:    }
316:    T sum = 0.0;
317:    for (std::size_t i = 0; i < rows; ++i) { sum += this->operator()(i, i); }
318:    return sum;
319:  }
320:  T Tr() const { return Trace(); }
321:
322:  void Print(const std::string &name = "") const {
323:    std::cout << "MMatrix::Print: " << name << " [" << rows << " x " << cols << "]" << std::endl;
324:    std::cout << std::setprecision(4);
325:    for (std::size_t i = 0; i < rows; ++i) {
326:      for (std::size_t j = 0; j < cols; ++j) { std::cout << this->operator()(i, j) << "\t"; }
327:      std::cout << std::endl;
328:    }
329:  }
330:
331:  // Size operators
```



```
332:  std::size_t size_row() const { return rows; }
333:  std::size_t size_col() const { return cols; }
334:
335: private:
336:  // Copy data from a to *this (after ReSize)
337:  void Copy(const MMatrix &a) {
338:    T *p = data + rows * cols;
339:    T *q = a.data + rows * cols;
340:    while (p > data) { *--p = *--q; }
341:  }
342:
343:  // Re-Allocate
344:  void ReSize(std::size_t r, std::size_t c) {
345:    if (data != nullptr) { delete[] data; }
346:    rows = r;
347:    cols = c;
348:    data = new T[rows * cols];
349:  }
350:
351:  std::size_t rows;
352:  std::size_t cols;
353:  T *         data;
354: };
355:
356:
357: }  // namespace gra
358:
359: #endif
```

./include/Graniitti/MTransport.h 1/1

```
// Optimal Transport Class
//
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

#ifndef MTRANSPORT_H
#define MTRANSPORT_H

// C++
#include <algorithm>
#include <complex>
#include <random>
#include <stdexcept>
#include <vector>

// Eigen
#include <Eigen/Dense>

// Own
#include "Graniitti/MMatrix.h"
#include "Graniitti/MTensor.h"

namespace gra {
namespace opt {

void ConvKernel(std::size_t n, std::size_t m, double lambda, MMatrix<double>& K);
void GibbsKernel(double lambda, const MMatrix<double>& C, MMatrix<double>& K);

double SinkHorn(MMatrix<double>& P, const MMatrix<double>& K, std::vector<double>& p,
                std::vector<double>& q, std::size_t iter);

}  // namespace opt
}  // namespace gra

#endif
```

./include/Graniitti/MNeuroJacobian.h 1/6

```
// GRANIITTI - Monte Carlo event generator for high energy diffraction
// https://github.com/mieskolainen/graniitti
//
// <NeuroJacobian neural net prototype>
//
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

#ifndef MNEUROJACOBIAN_H
#define MNEUROJACOBIAN_H

// C++ includes
#include <future>
#include <iostream>

// Own
#include <Graniitti/MAux.h>
#include <Graniitti/MH2.h>
#include <Graniitti/MMath.h>
#include <Graniitti/MProcess.h>
#include <Graniitti/MRandom.h>

// Eigen includes
#include <Eigen/Core>

// L-BFGS algorithms
#include <LBFGS/include/LBFGS.h>

// autodiff library
#include <autodiff/forward.hpp>
#include <autodiff/forward/eigen.hpp>
#include <autodiff/reverse.hpp>
#include <autodiff/reverse/eigen.hpp>

using namespace gra;
using gra::aux::indices;
using gra::math::PI;
using gra::math::pow2;

using Eigen::VectorXd;
using namespace Eigen;
using namespace autodiff;
using namespace LBFGSpp;

namespace gra {
namespace neurojac {
MRandom      randx;
unsigned int BATCHSIZE   = 0;
bool         FLAT_TARGET = false;

// declare a pointer to member function
MProcess *procptr;
double (MProcess::*ptfptr)(const std::vector<double> &randvec,
                           AuxIntData &               aux) = &MProcess::EventWeight;

// Function to be integrated
double func(std::vector<double> &x) {
  if (FLAT_TARGET) { return 1.0; }

  gra::AuxIntData aux;
  aux.vegasweight  = 1.0;
  aux.burn_in_mode = false;
  double y         = (gra::neurojac::procptr->*gra::neurojac::ptfptr)(x, aux);

  // Numerical protection
  if (std::isnan(y) || std::isinf(y)) { y = 0.0; }
  return y;
}

// Exponential linear unit
inline dual elu(dual z) {
  if (z >= 0.0) {
    return z;
  } else {
    return exp(z) - 1.0;
  }
}

// Hyperbolic functions
inline dual hypersinh(dual z) { return (exp(z) - exp(-z)) / 2; }
inline dual hypertanh(dual z) { return (exp(2.0 * z) - 1.0) / (exp(2.0 * z) + 1.0); }
```

./include/Graniitti/MNeuroJacobian.h 2/6

```
// Sigmoid
inline dual sigmoid(dual z) { return 1.0 / (1.0 + exp(-z)); }
// Modified Sigmoid
inline dual logexp(dual z) {
  const double alpha = 25.0;
  return 1.0 / alpha * log((1.0 + exp(alpha * z)) / (1.0 + exp(alpha * (z - 1.0))));
}

// Gaussian
dual gausprob(const VectorXdual &z, double mu, double sigma) {
  dual prod = 1.0;
  for (int i = 0; i < z.size(); ++i) {
    prod *= (1.0 / sqrt(2 * PI * sigma * sigma)) * exp(-(z[i] - mu) * (z[i] - mu) / 2.0);
  }
  return prod;
}

// Feedforward Network layer
class Layer {
 public:
  Layer(int M, int N) {
    w = MatrixXdual(M, N);
    b = VectorXdual(M);
  }
  ~Layer() {}

  MatrixXdual w;
  VectorXdual b;
};

// Parameter struct
struct NetParams {
  std::vector<Layer> L;

  // Return number of parameters
  int size_param() {
    int k = 0;
    for (std::size_t l = 0; l < L.size(); ++l) { k += L[l].w.rows() * L[l].w.cols(); }
    for (std::size_t l = 0; l < L.size(); ++l) { k += L[l].b.size(); }
    return k;
  }

  int  D = 0;  // Integral dimension
  dual alpha;
};

// Global parameters
NetParams par;

// Neural network function G: D^N -> D^N, D = [0 ... 1]
//
// This depends on global parameter struct p
//
//
VectorXdual G_net(const VectorXdual &x) {
  const double beta = 0.5;

  // Network input
  VectorXdual in(x.size());
  for (std::size_t i = 0; i < (unsigned int)x.size(); ++i) { in[i] = x[i]; }

  // Network layers
  for (std::size_t l = 0; l < par.L.size() - 1; ++l) {
    VectorXdual out(par.L[l].w.rows());
    for (std::size_t i = 0; i < (unsigned int)par.L[l].w.rows(); ++i) {
      for (std::size_t j = 0; j < (unsigned int)par.L[l].w.cols(); ++j) {
        out[i] += par.L[l].w(i, j) * in[j];
      }
      out[i] = hypertanh(out[i] + par.L[l].b[i]) * beta + (1.0 - beta) * out[i];
    }
    in = out;  // Next layer input is this layer output
  }

  // Output layer compression to [0,1] range
  const unsigned int last = par.L.size() - 1;
  VectorXdual        out(par.L[last].w.rows());
  for (std::size_t i = 0; i < (unsigned int)par.L[last].w.rows(); ++i) {
    for (std::size_t j = 0; j < (unsigned int)par.L[last].w.cols(); ++j) {
      out[i] += par.L[last].w(i, j) * in[j];
    }
    out[i] = logexp(out[i] + par.L[last].b[i]);
  }
  return out;
```

./include/Graniitti/MNeuroJacobian.h 3/6

```
}

class MNeuroJacobian {
 public:
  MNeuroJacobian() {}
  ~MNeuroJacobian() {}

  // Vector format to parameters
  static void Vec2Par(const std::vector<double> &w, NetParams par) {
    std::size_t k = 0;
    for (std::size_t l = 0; l < par.L.size(); ++l) {
      for (std::size_t i = 0; i < (unsigned int)par.L[l].w.rows(); ++i) {
        for (std::size_t j = 0; j < (unsigned int)par.L[l].w.cols(); ++j) {
          par.L[l].w(i, j) = w[k];
          ++k;
        }
      }
    }

    for (std::size_t l = 0; l < par.L.size(); ++l) {
      for (std::size_t j = 0; j < (unsigned int)par.L[l].b.size(); ++j) {
        par.L[l].b(j) = w[k];
        ++k;
      }
    }
  }

  // Random init for parameter vector
  std::vector<double> RandomInit(NetParams &par, double sigma) {
    std::vector<double> w(par.size_param());
    unsigned int        k = 0;

    // Connection matrices with random initialization
    for (std::size_t l = 0; l < par.L.size(); ++l) {
      for (std::size_t i = 0; i < (unsigned int)par.L[l].w.rows(); ++i) {
        for (std::size_t j = 0; j < (unsigned int)par.L[l].w.cols(); ++j) {
          w[k] = randx.G(0, sigma) * sqrt(2.0 / par.L[l].w.cols());  // He-et all style
          ++k;
        }
      }
    }
    // Bias initialized to zero
    for (std::size_t l = 0; l < par.L.size(); ++l) {
      for (std::size_t i = 0; i < (unsigned int)par.L[l].b.size(); ++i) {
        w[k] = 0.0;
        ++k;
      }
    }
    return w;
  }

  // Kullback-Leibler divergence loss function
  //
  //
  // For KL-divergence and algorithms with Jacobians, see:
  // [REFERENCE: Leow et al., Statistical Properties of Jacobian Maps ..., 2007]
  // http://www-sop.inria.fr/asclepios/cours/MVA/Module2/Papers/Leow_TMI07_LogUnbiased.pdf
  //
  // For neural networks and Jacobians, see:
  // [REFERENCE: Dillon et al, Tensorflow Bijector API,
  // https://arxiv.org/abs/1711.10604v1]
  //
  // Matrix derivative results, see:
  // [REFERENCE: M. Giles, http://eprints.maths.ox.ac.uk/1079/1/NA-08-01.pdf]
  //
  // MC Integration:
  // [REFERENCE: J. Bendavid, https://arxiv.org/pdf/1707.00028.pdf]
  //
  //
  static double singleloss(VectorXdual &z, const NetParams &par) {
    // Input likelihood value
    double p = val(gausprob(z, 0, 1));

    // Evaluate network
    VectorXdual u = G_net(z);

    // Jacobian matrix du/dz [AUTODIFF]
    MatrixXdual J = jacobian(G_net, u, z);

    // Absolute Jacobian determinant
    double absDetJ = abs(val(J.determinant()));

    // ============================================================
```

./include/Graniitti/MNeuroJacobian.h 4/6

```
250:     // Evaluate integrand function
251:     std::vector<double> u_(u.size());
252:     for (std::size_t i = 0; i < (unsigned int)u.size(); ++i) { u_[i] = val(u[i]); }
253:     double fG = func(u_);
254:     // ============================================================
255:
256:     // Prior term
257:     double KL = 0.0;
258:     if (p > 0) { KL = log(p); }
259:
260:     // Jacobian determinant term
261:     if (absDetJ > 0) { KL = KL - log(absDetJ); }
262:
263:     // Target function fidelity term
264:     if (fG > 0) { KL = KL - log(fG); }
265:
266:     return KL;
267:   }
268:
269:   static dual loss(VectorXdual &Z, const NetParams &par) {
270:     VectorXdual z(par.D);
271:     dual        sumloss = 0.0;
272:
273:     // Loop over all samples
274:     int k = 0;
275:     while (true) {
276:       // Pick prior p(z) distribution samples
277:       for (std::size_t i = 0; i < (unsigned int)par.D; ++i) {
278:         z[i] = Z[k];
279:         ++k;
280:       }
281:       sumloss = sumloss + singleloss(z, par);
282:       if (k == Z.size()) break;
283:     }
284:     return sumloss;
285:   }
286:
287:   // Evaluate loss and gradient vector
288:   static double CostGrad(const std::vector<double> &w, std::vector<double> &gradv) {
289:     // Generate new virtual batch from noise distribution (prior)
290:     VectorXdual Z(BATCHSIZE * par.D);
291:     for (std::size_t i = 0; i < (unsigned int)Z.size(); ++i) { Z[i] = randx.G(0, 1); }
292:
293:     // Update vector to global parameters
294:     Vec2Par(w, par);
295:
296:     // ------------------------------------------------------------------
297:     // Evaluate gradient vector components numerically (not very optimal..)
298:     std::size_t k = 0;
299:     gradv.resize(w.size());
300:
301:     // Numerical derivative epsilon
302:     const double h = 1e-4;
303:
304:     for (std::size_t l = 0; l < par.L.size(); ++l) {
305:       for (std::size_t i = 0; i < (unsigned int)par.L[l].w.rows(); ++i) {
306:         for (std::size_t j = 0; j < (unsigned int)par.L[l].w.cols(); ++j) {
307:           par.L[l].w(i, j)   = w[k] + h;
308:           const double f_pos = val(loss(Z, par));
309:           par.L[l].w(i, j)   = w[k] - h;
310:           const double f_neg = val(loss(Z, par));
311:
312:           gradv[k] = (f_pos - f_neg) / (2 * h);
313:
314:           par.L[l].w(i, j) = w[k];  // back to previous value
315:           ++k;
316:         }
317:       }
318:     }
319:
320:     for (std::size_t l = 0; l < par.L.size(); ++l) {
321:       for (std::size_t j = 0; j < (unsigned int)par.L[l].b.size(); ++j) {
322:         par.L[l].b(j)      = w[k] + h;
323:         const double f_pos = val(loss(Z, par));
324:         par.L[l].b(j)      = w[k] - h;
325:         const double f_neg = val(loss(Z, par));
326:
327:         gradv[k]      = (f_pos - f_neg) / (2 * h);
328:         par.L[l].b(j) = w[k];  // back to previous value
329:         ++k;
330:       }
331:     }
332:
```

./include/Graniitti/MNeuroJacobian.h 5/6

```
333:     const double cost = val(loss(Z, par));
334:     std::cout << "loss = " << cost << std::endl;
335:     return cost;
336:   }
337:
338:   // Naive gradient descent
339:   void NaiveGradDesc(std::vector<double> &w, unsigned int MAXITER, double rate) {
340:     // ------------------------------------------------------------------------------
341:     // Gradient Descent
342:     // w_n+1 = w_n - gamma*Grad(F(w_n))
343:     unsigned int iter = 0;
344:
345:     // gradvient vector
346:     std::vector<double> gradv(w.size(), 0.0);
347:
348:     while (true) {
349:       CostGrad(w, gradv);
350:       // Naive gradient descent update
351:       for (std::size_t k = 0; k < gradv.size(); ++k) {
352:         w[k] = w[k] - rate * gradv[k];
353:         printf("w[%3lu] = %6.3f    <gradv[%3lu] = %10.3E> \n", k, w[k], k, gradv[k]);
354:       }
355:       ++iter;
356:
357:       if (iter > MAXITER) break;
358:     }
359:   }
360:
361:   // Test network dimensions
362:   void TestNetworkDimensions() {
363:     for (std::size_t l = 0; l < par.L.size() - 1; ++l) {
364:       if (par.L[l + 1].w.cols() != par.L[l].w.rows()) {
365:         throw std::invalid_argument("Network layer dimension mismatch!");
366:       }
367:     }
368:   }
369:
370:   // Optimize network
371:   void Optimize() {
372:     // ------------------------------------------------------------------
373:     // Random init (too high sigma -> may get trapped to a bad local minima)
374:     double             sigma = 1e-4;
375:     std::vector<double> w     = RandomInit(par, sigma);
376:
377:     // ------------------------------------------------------------------
378:     // By Jacobi formula: d/dt ln det A(t) = tr[A(t)^{-1} d/dt A]
379:     // ------------------------------------------------------------------
380:
381:     const unsigned int n = w.size();
382:
383:     LBFGSParam<double> param;
384:
385:     // Initial guess
386:     VectorXd x = VectorXd::Zero(n);
387:     for (std::size_t i = 0; i < n; ++i) { x[i] = w[i]; }
388:
389:     // Function object
390:     Neurocost fun(n);
391:
392:     // x will be overwritten to be the best point found
393:     double fx;
394:     int    niter;
395:
396:     FLAT_TARGET = true;
397:     std::cout << "Solving flat target: " << std::endl;
398:
399:     param.epsilon        = 1e-4;
400:     param.max_iterations = 10;
401:
402:     LBFGSSolver<double> solver0(param);
403:     niter = solver0.minimize(fun, x, fx);
404:
405:     FLAT_TARGET = false;
406:     std::cout << "Solving real target: " << std::endl;
407:
408:     param.epsilon        = 1e-4;
409:     param.max_iterations = 20;
410:
411:     LBFGSSolver<double> solver1(param);
412:     niter = solver1.minimize(fun, x, fx);
413:
414:     std::cout << niter << " iterations" << std::endl;
415:     std::cout << "x = \n" << x.transpose() << std::endl;
```

./include/Graniitti/MNeuroJacobian.h 6/6

```
416:     std::cout << "f(x) = " << fx << std::endl;
417:   }
418:
419:  private:
420:   class Neurocost {
421:    public:
422:     Neurocost(int _n) { n = _n; }
423:
424:     double fx = 0.0;
425:     double operator()(const VectorXd &x, VectorXd &grad) {
426:       // Map to vectors
427:       std::vector<double> w(x.size(), 0.0);
428:       std::vector<double> gradv(x.size(), 0.0);
429:
430:       for (std::size_t i = 0; i < (unsigned int)x.size(); ++i) { w[i] = x[i]; }
431:
432:       fx = CostGrad(w, gradv);
433:
434:       for (std::size_t i = 0; i < (unsigned int)x.size(); ++i) { grad[i] = gradv[i]; }
435:
436:       return fx;
437:     }
438:
439:    private:
440:     int n;
441:   };
442:
443: };  // MNeuroJacobian
444:
445: }  // Namespace neurojac
446: }  // Namespace gra
447:
448: #endif
```

./include/Graniitti/MAux.h 1/5

```
  1: // I/O aux functions
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: #ifndef MAUX_H
  7: #define MAUX_H
  8:
  9: // C++
 10: #include <limits.h>
 11: #include <unistd.h>
 12: #include <complex>
 13: #include <mutex>
 14: #include <random>
 15: #include <regex>
 16: #include <sstream>
 17: #include <stdexcept>
 18: #include <vector>
 19: //#include <experimental/filesystem>
 20:
 21: // Own
 22: #include "Graniitti/M4Vec.h"
 23: #include "Graniitti/MMatrix.h"
 24:
 25: // LHAPDF
 26: #include "LHAPDF/LHAPDF.h"
 27:
 28: // HepMC3
 29: #include "HepMC3/FourVector.h"
 30:
 31: // Eigen
 32: #include <Eigen/Dense>
 33:
 34: // Libraries
 35: #include "rang.hpp"
 36:
 37: namespace gra {
 38: namespace aux {
 39: // Example:
 40: //
 41: // @SOMECOMMAND:value
 42: // @PDG[992](M:300, W:0)
 43:
 44: // Format options:
 45: //
 46: // @id:value is represented with map<"_SINGLET_", value>
 47: //
 48: // @id:value
 49: // @id[arg]
 50: // @id[!]arg]
 51: // @id[target1,target2,...]{arg}
 52:
 53: struct OneCMD {
 54:   std::string                id;
 55:   std::vector<std::string>   target;
 56:   std::map<std::string, std::string> arg;
 57:
 58:   void Print() {
 59:     std::cout << "id : " << id << std::endl;
 60:
 61:     std::cout << "[target] : [";
 62:     for (std::size_t i = 0; i < target.size(); ++i) {
 63:       std::cout << target[i];
 64:       if (i < target.size() - 1) { std::cout << ","; }
 65:     }
 66:     std::cout << "] " << std::endl;
 67:
 68:     std::cout << "{arg} : " << std::endl;
 69:     for (const auto &x : arg) { std::cout << x.first << ":" << x.second << std::endl; }
 70:     std::cout << std::endl;
 71:   }
 72: };
 73:
 74: // M4Vec to HepMC3::FourVector
 75: inline HepMC3::FourVector M4Vec2HepMC3(const M4Vec &v) {
 76:   return HepMC3::FourVector(v.X(), v.Y(), v.Z(), v.E());
 77: }
 78: // HepMC3::FourVector to M4Vec
 79: inline M4Vec HepMC2M4Vec(const HepMC3::FourVector &v) { return M4Vec(v.x(), v.y(), v.z(), v.e()); }
 80:
 81: // Eigen to std::vector
 82: inline std::vector<double> Eigen2Vector(const Eigen::VectorXd &x) {
 83:   std::vector<double> y(x.size());
```

```
   for (int i = 0; i < x.size(); ++i) { y[i] = x[i]; }
   return y;
}

// std::vector to Eigen
template <typename T>
inline Eigen::VectorXd Vector2Eigen(const std::vector<T> &x) {
   Eigen::VectorXd y(x.size());
   for (std::size_t i = 0; i < x.size(); ++i) { y[i] = x[i]; }
   return y;
}

// MMatrix to Eigen matrix
template <typename T>
inline Eigen::MatrixXd Matrix2Eigen(const MMatrix<T> &M) {
   Eigen::MatrixXd eM(M.size_row(), M.size_col());
   for (std::size_t i = 0; i < M.size_row(); ++i) {
      for (std::size_t j = 0; j < M.size_col(); ++j) { eM(i, j) = M[i][j]; }
   }
   return eM;
}

// System information
void        PrintArgv(int argc, char *argv[]);
void        AutoDownloadLHAPDF(const std::string pdfname);
std::string ExecCommand(const std::string &cmd);
std::string GetExecutablePath();
std::string GetBasePath(std::size_t level);

/*
std::string GetCurrentPath();
bool FileExist(const std::experimental::filesystem::path& p,
               std::experimental::filesystem::file_status s =
std::experimental::filesystem::file_status{});
*/

std::uintmax_t GetFileSize(const std::string &filename);
void           GetProcessMemory(double &peak_use, double &resident_use);
void           GetDiskUsage(const std::string &path, int64_t &size, int64_t &free, int64_t &used);
unsigned long long TotalSystemMemory();
std::string        SystemName();
std::string        HostName();
const std::string  DateTime();

// Progress bar
void PrintProgress(double ratio);
void ClearProgress();

// djb2hash function
unsigned long djb2hash(const std::string &s);

// Simple CSV reader
void ReadCSV(const std::string &inputfile, std::vector<std::vector<std::string>> &output);

// Input processing
std::string GetInputData(const std::string &inputfile);

bool IsIntegerDigits(const std::string &str);

// Trim extra spaces of a string
void TrimExtraSpace(std::string &value);
void TrimLeadSpace(std::string &value);
void TrimTrailSpace(std::string &value);
void TrimEmptySpace(std::string &value);
void TrimAllSpace(std::string &value);

// Number to string with formatting
template <typename T>
std::string ToString(const T value, const unsigned int n = 6) {
   std::ostringstream out;
   out.precision(n);
   out << std::fixed << value;
   return out.str();
}

// Quantum numbers as a string
std::string ParityToString(int value);
std::string Charge3XtoString(int q3);
std::string Spin2XtoString(int J2);

// String splitting
std::vector<std::string> SplitStr2Str(std::string input, const char delim = ',',
```



```
                                      bool trimextraspace = true);
std::vector<int>         SplitStr2Int(std::string input, const char delim = ',');
std::vector<std::string> Extract(const std::string &str);

// Split string to int or double
template <class T>
std::vector<T> SplitStr(std::string input, T type, const char delim = ',') {
  std::vector<T>    output;
  std::stringstream ss(input);

  // Get inputfiles by comma
  while (ss.good()) {
    std::string substr;
    std::getline(ss, substr, delim);

    TrimExtraSpace(substr);

    // Detect type >>
    // int
    if (std::is_same<T, int>::value) {
      output.push_back(std::stoi(substr));
    }
    // double
    else if (std::is_same<T, double>::value) {
      output.push_back(std::stod(substr));
    }
  }
  return output;
}

// Check if file exists
bool FileExist(const std::string &name);

void PrintNotice();
void PrintWarning();
void PrintGameOver();
void PrintFlashScreen(rang::fg pcolor);
void PrintVersion();

// Bar print
void PrintBar(std::string str, unsigned int N = 74);

// Create directory
void CreateDirectory(std::string fullpath);

// Version information
void CheckUpdate();
void CreateVersionJSON();

double      GetVersion();
std::string GetVersionType();
std::string GetVersionDate();
std::string GetVersionUpdate();
std::string GetVersionString();
std::string GetVersionTLatex();
std::string GetWebTLatex();

// Assert functions


// Check cut, lower value needs to be smaller than upper value
template <typename T>
bool AssertCut(std::vector<T> cut, const std::string &name = "", bool dothrow = false) {
  if (cut.size() != 2) {
    throw std::invalid_argument("AssertCut: Input '" + name + "' vector size not 2");
  }
  if (cut[1] <= cut[0]) {
    if (dothrow) {
      std::string message = "AssertCut: Input '" + name + "' with [" + std::to_string(cut[0]) +
                            "," + std::to_string(cut[1]) +
                            "] (maximum value smaller than minimum value)";
      throw std::invalid_argument(message);
    }
    return false;
  }
  return true;
}


// Check cut obeys given boundaries
template <typename T>
bool AssertCutRange(std::vector<T> cut, std::vector<T> bounds, const std::string &name = "",
                    bool dothrow = false) {
```



```
  if (cut.size() != 2) {
    throw std::invalid_argument("AssertCutRange: Input '" + name + "' vector size not 2");
  }
  if (!AssertCut(cut, name, dothrow)) { return false; }

  if (cut[0] < bounds[0] || cut[1] > bounds[1]) {
    if (dothrow) {
      std::string message = "AssertCutRange: Input '" + name + "' with [" + std::to_string(cut[0]) +
                            "," + std::to_string(cut[1]) + "]" + " invalid given bounds: [" +
                                  std::to_string(bounds[0]) + "," + std::to_string(bounds[1]) + "]";
      throw std::invalid_argument(message);
    }
    return false;
  }
  return true;
}

// Assert compare function, threshold 0.01 means 1 percent accuracy
template <typename T>
bool AssertRatio(T value, T reference, T threshold, const std::string &name = "",
                 bool dothrow = false) {
  bool   ok    = false;
  const T ratio = value / reference;
  if (ratio < (1.0 + threshold) && ratio > (1.0 - threshold)) { ok = true; }
  if (!ok && dothrow) {
    throw std::invalid_argument("AsserRatio: Input '" + name + "' = " + std::to_string(value) +
                                " not within reference = " + std::to_string(reference) +
                                " under threshold = " + std::to_string(threshold));
  }
  return ok;
}

// Assert range [a,b]
template <typename T>
bool AssertRange(T value, std::vector<T> range, const std::string &name = "",
                 bool dothrow = false) {
  if (range.size() != 2) {
    throw std::invalid_argument("AssertRange: Input '" + name + "' , range vector size is not 2!");
  }
  bool ok = false;
  if (value >= range[0] && value <= range[1]) { ok = true; }
  if (!ok && dothrow) {
    throw std::invalid_argument("AssertRange: Input '" + name + "' = " + std::to_string(value) +
                                " out of range [" + std::to_string(range[0]) + "," +
                                std::to_string(range[1]) + "]");
  }
  return ok;
}

// Assert value if found from a set of numbers
template <typename T>
bool AssertSet(T value, std::vector<T> set, const std::string &name = "", bool dothrow = false) {
  bool ok = false;
  for (const auto &i : set) {
    if (value == i) {
      ok = true;
      break;
    }
  }
  if (!ok && dothrow) {
    throw std::invalid_argument("AssertSet: Input '" + name + "' = " + std::to_string(value) +
                                " not found from the input set");
  }
  return ok;
}

// ------------------------------------------------------------------
// Templates for enhanced index based looping
// for (const auto& i : indices(vector)) { vector[i] = foo; ... }

template <typename T>
struct index_range {
  struct iterator {
    bool     operator!=(iterator x) const { return index != x.index; }
    iterator operator++() {
      ++index;
      return *this;
    }
    T operator*() const { return index; }

    T index;
  };
```



```
  iterator begin() const { return {0}; }
  iterator end() const { return {n}; }

  T n;
};

template <typename T, typename Index = typename T::size_type>
index_range<Index> indices(const T &container) {
  return {container.size()};
}
// ------------------------------------------------------------------

std::vector<OneCMD>      SplitCommands(const std::string &fullstr);
std::vector<std::size_t> FindOccurance(const std::string &str, const std::string &sub);

}  // namespace aux
}  // namespace gra

#endif
```

./include/Graniitti/MFFT.h 1/1

```
// Simple in-place Fast Fourier Transform with radix-2 using std::valarray
//
// Normalization is 1 to 1, that is: ifft( fft(x) ) = x
//
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

#ifndef MFFT_H
#define MFFT_H

// C++
#include <complex>
#include <valarray>

namespace gra {
#ifndef M__PI
#define M__PI 3.14159265358979323846264338327950288419716939937510582097494 4L
#endif

namespace MFFT {
// In-place Cooley$\200\223Tukey FFT
template <typename T>
void fft(std::valarray<std::complex<T>> &x) {
  const std::size_t N = x.size();
  if (N <= 1) { return; }  // Trivial case/recursion ends, X = x
  if ((N & (N - 1)) != 0) {
    throw std::invalid_argument("ERROR: MFFT::fft: Input x.size() = " + std::to_string(N) +
                                " not a power of 2!");
  }

  // Radix-2 step
  std::valarray<std::complex<T>> E = x[std::slice(0, N / 2, 2)];
  std::valarray<std::complex<T>> O = x[std::slice(1, N / 2, 2)];

  // Even and odd part via recursion
  MFFT::fft(E);
  MFFT::fft(O);

  for (std::size_t k = 0; k < N / 2; ++k) {
    const std::complex<T> t = std::exp(std::complex<T>(0, -2.0 * M__PI * k / N));
    x[k]     = E[k] + O[k] * t;
    x[k + N / 2] = E[k] - O[k] * t;
  }
}

// In-place Cooley-Tukey IFFT
template <typename T>
void ifft(std::valarray<std::complex<T>> &x) {
  x = x.apply(std::conj);  // Conjugate
  MFFT::fft(x);            // FFT
  x = x.apply(std::conj);  // Conjugate
  x /= x.size();           // Normalize
}
}  // namespace MFFT

}  // gra namespace ends

#endif
```

./include/Graniitti/MGraniitti.h 1/4

```
// GRANIITTI Monte Carlo main class
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

#ifndef MGRANIITTI_H
#define MGRANIITTI_H

// C++
#include <complex>
#include <mutex>
#include <random>
#include <thread>
#include <vector>

// HepMC3
#include "HepMC3/GenEvent.h"
#include "HepMC3/WriterAscii.h"
#include "HepMC3/WriterAsciiHepMC2.h"
#include "HepMC3/WriterHEPEVT.h"

// Own
#include "Graniitti/M4Vec.h"
#include "Graniitti/MAux.h"
#include "Graniitti/MContinuum.h"
#include "Graniitti/MEikonal.h"
#include "Graniitti/MFactorized.h"
#include "Graniitti/MKinematics.h"
#include "Graniitti/MMatrix.h"
#include "Graniitti/MParton.h"
#include "Graniitti/MProcess.h"
#include "Graniitti/MQuasiElastic.h"
#include "Graniitti/MSpin.h"
#include "Graniitti/MTimer.h"
#include "Graniitti/MVEGAS.h"

namespace gra {

// Simple MC parameters
struct MCPARAM {
  double       PRECISION  = 0.05;     // Integral relative precision
  unsigned int MIN_EVENTS = 1000000;  // Minimum number of events to be sampled
};

// Integration statistics
class Stats {
 public:
  void Accumulate(const gra::AuxIntData &aux) {
    evaluations += 1.0;

    amplitude_ok += (aux.amplitude_ok ? 1.0 : 0.0);
    kinematics_ok += (aux.kinematics_ok ? 1.0 : 0.0);
    fiducts_ok += (aux.fiducts_ok ? 1.0 : 0.0);
    vetocuts_ok += (aux.vetocuts_ok ? 1.0 : 0.0);
  }

  // Calculate integrated cross section sigma and its
  // error for direct (simple) sampling. NOT TO BE USED WITH VEGAS!
  void CalculateCrossSection() {
    // <f>
    sigma = Wsum / evaluations;
    // err^2 = <f^2> - <f>^2
    sigma_err2 = W2sum / evaluations - gra::math::pow2(sigma);
    // err_trials = err^2 / sqrt(trials) (standard error of the mean)
    sigma_err = gra::math::msqrt(sigma_err2 / evaluations);
  }

  double amplitude_ok  = 0.0;
  double kinematics_ok = 0.0;
  double fiducts_ok    = 0.0;
  double vetocuts_ok   = 0.0;

  // Keep as double to avoid overflow of range
  double evaluations = 0.0;  // Integrand evaluations
  double trials      = 0.0;  // Event generation trials

  unsigned int generated  = 0.0;  // Event generation
  unsigned int N_overflow = 0.0;  // Weight overflows

  // Cross section and its error
  double sigma      = 0.0;
  double sigma_err  = 0.0;
  double sigma_err2 = 0.0;
```

./include/Graniitti/MGraniitti.h 2/4

```

  // Weight statistics FLAT MC
  double Wsum  = 0.0;
  double W2sum = 0.0;
  double maxW  = 0.0;

  // Weight statistics VEGAS MC
  double maxf = 0.0;
  double chi2 = 0.0;
};

class MGraniitti {
 public:
  // Destructor & Constructor
  MGraniitti();
  ~MGraniitti();

  // For cross-section for HepMC3 output
  void ForceXS(double xs) { xsforced = xs; }

  // Read parameters
  void ReadInput(const std::string &inputfile, const std::string cmd_PROCESS = "null");

  // Initialize memory
  void InitProcessMemory(std::string process, unsigned int RNDSEED);
  void InitMultiMemory();

  // Set simple MC parameters
  void SetMCParam(MCPARAM &in);

  // Set VEGAS parameters
  void SetVegasParam(const VEGASPARAM &in);

  // Set external file handle
  void SetHepMC2Output(std::shared_ptr<HepMC3::WriterAsciiHepMC2> &hepmc,
                       const std::string &                        OUTPUTNAME) {

    OUTPUT       = OUTPUTNAME;
    FORMAT       = "hepmc2";
    outputHepMC2 = hepmc;
  }
  void SetHepMC3Output(std::shared_ptr<HepMC3::WriterAscii> &hepmc, const std::string &OUTPUTNAME) {
    OUTPUT       = OUTPUTNAME;
    FORMAT       = "hepmc3";
    outputHepMC3 = hepmc;
  }

  // Maximum weight set/get
  void   SetMaxweight(double w);
  double GetMaxweight() const;

  // Get number of CPU cores
  void SetCores(int N) {
    CORES = N;
    if (CORES == 0) {  // SETUP number of threads automatically
      CORES = std::round(std::thread::hardware_concurrency() * 1.5);

      // If autodetection fails, set 1
      if (CORES < 1) { CORES = 1; }
    }
    if (CORES < 0) {
      std::string str = "MGraniitti::SetCORES: CORES < 0";
      throw std::invalid_argument(str);
    }
  }
  int  GetCores() const { return CORES; }
  void SetIntegrator(const std::string &integrator) { INTEGRATOR = integrator; }
  void SetWeighted(bool weighted) { WEIGHTED = weighted; }
  // Output file name
  void SetOutput(const std::string &output) { OUTPUT = output; }
  // Output file format
  void SetFormat(const std::string &format) {
    if (format == "hepmc3" || format == "hepmc2" || format == "hepevt") {
      FORMAT = format;
    } else {
      throw std::invalid_argument("MGraniitti::SetFormat: Unknown output format: " + format +
                                  " (valid: hepmc3, hepmc2, hepevt)");

    }
  }

  // Special method for combining histograms from each thread
  void HistogramFusion();

  // Get cross section and error
```

./include/Graniitti/MGraniitti.h 3/4

```
  void GetXS(double &xs, double &xs_err) const {
    xs     = stat.sigma;
    xs_err = stat.sigma_err;
  }

  // Set number of (un)weighted events to be generated
  void SetNumberOfEvents(int n) {
    if (n < 0) {
      throw std::invalid_argument(
          "MGraniitti::SetNumberOfEvents():: Error: Number "
          "of events < 0!");
    }
    NEVENTS = n;
  }
  int GetNumberOfEvents() const { return NEVENTS; }

  // Return processes
  std::vector<std::string> GetProcessNumbers() const;

  // Initialize generator
  void Initialize();
  void Initialize(const MEikonal &eikonal_in);

  // Process object pointer, public so methods can be accessed
  MProcess *proc = nullptr;

  void PrintHistograms();
  void ReadGeneralParam(const std::string &inputfile);
  void ReadProcessParam(const std::string &inputfile, const std::string cmd_PROCESS = "null");
  void ReadIntegralParam(const std::string &inputfile);
  void ReadGenCuts(const std::string &inputfile);
  void ReadFidCuts(const std::string &inputfile);
  void ReadVetoCuts(const std::string &inputfile);
  void ReadModelParam(const std::string &inputfile) const;

  void Generate();

  std::string PROCESS = "null";  // Physics process identifier

  bool        WEIGHTED   = false;   // Unweighted or weighted event generation
  int         NEVENTS    = 0;       // Number of events to be generated
  int         CORES      = 0;       // Number of CPU cores (threads) in use
  std::string INTEGRATOR = "null";  // Integrator (VEGAS, FLAT, ...)

  // SILENT OUTPUT
  bool HILJAA = false;

 private:
  // ------------------------------------------------------------------------
  // Copy and assignment disabled by making the private
  MGraniitti(const MGraniitti &);
  MGraniitti &operator=(const MGraniitti &);
  // ------------------------------------------------------------------------

  // Integration statistics
  Stats stat;

  // Histogram fusion
  bool hist_fusion_done = false;

  // Input string
  std::string FULL_INPUT_STR = "null";

  // HepMC outputfile
  std::string FULL_OUTPUT_STR = "null";
  std::string OUTPUT          = "null";
  std::string FORMAT          = "null";  // hepmc3 or hepmc2 or hepevt

  std::shared_ptr<HepMC3::GenRunInfo>        runinfo      = nullptr;
  std::shared_ptr<HepMC3::WriterAscii>       outputHepMC3 = nullptr;
  std::shared_ptr<HepMC3::WriterAsciiHepMC2> outputHepMC2 = nullptr;
  std::shared_ptr<HepMC3::WriterHEPEVT>      outputHEPEVT = nullptr;

  // VEGAS creates copies here
  std::vector<MProcess *> pvec;
  MContinuum              proc_C;
  MFactorized             proc_F;
  MQuasiElastic           proc_Q;
  MParton                 proc_P;

  // Forced cross-section for HepMC3 output
  double xsforced = -1;
```

./include/Graniitti/MGraniitti.h 4/4

```
250:   // Global timing
251:   MTimer global_tictoc;
252:   MTimer local_tictoc;
253:   MTimer atime;
254:   double time_t0  = 0.0;
255:   double itertime = 0.0;
256:
257:   // ------------------------------------------------
258:   // Process MC integration generic variables
259:
260:   // Generation mode (integration = 0, event generation = 1)
261:   unsigned int GMODE = 0;
262:
263:   // ------------------------------------------------
264:   // FLAT MC
265:   MCPARAM mcparam;
266:
267:   // ------------------------------------------------
268:   // VEGAS MC
269:
270:   // Parameters
271:   VEGASPARAM vparam;
272:
273:   // DATA
274:   VEGASData VD;
275:
276:   void VEGASInit(unsigned int init, unsigned int calls);
277:   int  VEGAS(unsigned int init, unsigned int calls, unsigned int iter, unsigned int N);
278:   void VEGASMultiThread(unsigned int N, unsigned int tid, unsigned int init,
279:                         unsigned int LOCALcalls);
280:
281:   // ------------------------------------------------
282:
283:   void UnifyHistogramBounds();
284:
285:   // Calculate cross section
286:   void CalculateCrossSection();
287:
288:   // Vegas wrapper
289:   double VegasWrapper(std::vector<double> &randvec, double wgt);
290:
291:   // Event sampling/generation
292:   void CallIntegrator(unsigned int N);
293:   void SampleVegas(unsigned int N);
294:   void SampleFlat(unsigned int N);
295:   void SampleNeuro(unsigned int N);
296:
297:   // Helper functions
298:   void           PrintInit() const;
299:   int            SaveEvent(MProcess *pr, double W, double MAXW, const gra::AuxIntData &aux);
300:   void           PrintStatus(unsigned int events, unsigned int N, MTimer &tictoc, double timercut);
301:   void           PrintStatistics(unsigned int N);
302:   gra::PARAM_RES ReadFactorized(const std::string &resparam_str);
303:   void           InitFileOutput();
304:
305:   // Interpreter commands
306:   std::vector<aux::OneCMD> syntax;
307: };
308:
309: // Launcher functions
310: void MLaunch(MGraniitti gen, int randomseed, int tid, int events);
311: void MThreader(MGraniitti &gen);
312:
313: }  // namespace gra
314:
315: #endif
```

./include/Graniitti/Analysis/MROOT.h 1/4

```
  1: // ROOT style namespace
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: #ifndef MROOT_H
  7: #define MROOT_H
  8:
  9: #include <complex>
 10: #include <memory>
 11: #include <tuple>
 12: #include <vector>
 13:
 14: // Own
 15: #include "Graniitti/MMath.h"
 16:
 17: // ROOT
 18: #include "TCanvas.h"
 19: #include "TColor.h"
 20: #include "TLatex.h"
 21: #include "TPad.h"
 22: #include "TROOT.h"
 23: #include "TStyle.h"
 24:
 25: // Own
 26: #include "Graniitti/MAux.h"
 27:
 28: namespace gra {
 29: namespace rootstyle {
 30:
 31: // Returns a new colormap
 32: inline void CreateColorMap(std::vector<int>& color, std::vector<std::shared_ptr<TColor>>& rootcolor,
 33:                            int COLORSCHEME = 1) {
 34:   std::vector<std::vector<double>> colormap(150);
 35:
 36:   if (COLORSCHEME == 1) {
 37:     // "Modern colormap"
 38:     std::vector<std::vector<double>> cm = {{0, 0.4470, 0.7410},      {0.8500, 0.3250, 0.0980},
 39:                                            {0.9290, 0.6940, 0.1250}, {0.4940, 0.1840, 0.5560},
 40:                                            {0.4660, 0.6740, 0.1880}, {0.3010, 0.7450, 0.9330},
 41:                                            {0.6350, 0.0780, 0.1840}};
 42:     colormap                             = cm;
 43:   }
 44:
 45:   if (COLORSCHEME == 2) {
 46:     // "Classic colormap"
 47:     std::vector<std::vector<double>> cm = {{0, 0, 0.9},       {0, 0.5, 0},     {0.9, 0, 0},
 48:                                            {0, 0.75, 0.75},   {0.75, 0, 0.75}, {0.75, 0.75, 0},
 49:                                            {0.25, 0.25, 0.25}};
 50:     colormap                             = cm;
 51:   }
 52:
 53:   color     = std::vector<int>(colormap.size(), 0);
 54:   rootcolor = std::vector<std::shared_ptr<TColor>>(color.size(), nullptr);
 55:
 56:   for (const auto& i : aux::indices(color)) {
 57:     color[i] = TColor::GetFreeColorIndex();
 58:     // color[i] = 9000 + i;  // some big number not used
 59:
 60:     // ROOT style, we need to create some hidden memory part
 61:     rootcolor[i] =
 62:         std::make_shared<TColor>(color[i], colormap[i][0], colormap[i][1], colormap[i][2]);
 63:   }
 64: }
 65:
 66:
 67: // Create grid canvas with N subpads
 68: inline void AutoGridCanvas(std::shared_ptr<TCanvas>& c1, unsigned int N) {
 69:   unsigned int ADD = 0;
 70:   while (true) {  // Adjust grid size
 71:     const unsigned int val = std::sqrt(N + ADD);
 72:     if (val * val == (N + ADD)) { break; }
 73:     ++ADD;
 74:   }
 75:   // Calculate if we have a full empty row -> remove that
 76:   unsigned int DEL = 0;
 77:   if (ADD * ADD - ADD == N) { DEL = 1; }
 78:
 79:   const unsigned int COLS = std::sqrt(N + ADD);
 80:   const unsigned int ROWS = std::sqrt(N + ADD) - DEL;
 81:
 82:   // Adjust aspect ratio
 83:   if (ROWS == COLS) {
```

./include/Graniitti/Analysis/MROOT.h 2/4

```
 84:     c1 = std::make_shared<TCanvas>("c1", "c1", 700, 600);  // horizontal, vertical
 85:   }
 86:   if (ROWS != COLS) {
 87:     c1 = std::make_shared<TCanvas>("c1", "c1", 700, 450);  // horizontal, vertical
 88:   }
 89:
 90:   c1->Divide(COLS, ROWS, 0.002, 0.001);
 91:
 92:   // This is needed
 93:   gStyle->SetPadLeftMargin(0.15);
 94: }
 95:
 96: // CubeHelix colormap generator
 97: //
 98: // N     = number of discrete steps
 99: // start = start color (1 = red ... 2 = green ... 3 = red)
100: // R     = number of helix rotations
101: // hue   = hue, with 0 gives black&white
102: // gamma = intensity correction
103: //
104: // Default CubeHelix(256, 0.5, -1.5, 1.2, 1.0);
105: //
106: // [REFERENCE: D.A. Green, https://arxiv.org/abs/1108.5083]
107: // https://www.mrao.cam.ac.uk/~dag/CUBEHELIX
108: //
109: inline std::vector<std::vector<double>> CubeHelix(int N, double start, double R, double hue,
110:                                                   double gamma) {
111:   auto limitfunc = [](std::vector<double>& x) {
112:     for (std::size_t i = 0; i < x.size(); ++i) {
113:       if (x[i] < 0.0) { x[i] = 0.0; }
114:       if (x[i] > 1.0) { x[i] = 1.0; }
115:     }
116:   };
117:
118:   // Color matrix
119:   const std::vector<std::vector<double>> A = {
120:       {-0.14861, 1.78277}, {-0.29227, -0.90649}, {1.97294, 0}};
121:   const double PI = 3.14159265359;
122:
123:   // Steps, Red, Green, Blue
124:   std::vector<std::vector<double>> M(4, std::vector<double>(N, 0.0));
125:
126:   // Lightning
127:   const double maxlight  = 1.0;
128:   const double minlight  = 0.0;
129:   const double lightstep = (maxlight - minlight) / N;
130:
131:   for (int i = 1; i <= N; ++i) {
132:     // Rotation angle and shift
133:     double       alpha = (i - 1.0) / (N - 1.0);
134:     const double phi   = 2.0 * PI * (start / 3.0 + 1.0 + R * alpha);
135:
136:     // Apply gamma-correction
137:     alpha             = std::pow(alpha, gamma);
138:     const double a = hue * alpha * (1.0 - alpha) / 2.0;
139:
140:     // Affine Map
141:     const std::vector<double> x = {a * std::cos(phi), a * std::sin(phi)};
142:     std::vector<double>       y = {A[0][0] * x[0] + A[0][1] * x[1] + alpha,
143:                                    A[1][0] * x[0] + A[1][1] * x[1] + alpha,
144:                                    A[2][0] * x[0] + A[2][1] * x[1] + alpha};
145:     limitfunc(y);  // Limit values to [0,1]
146:
147:     // Save values
148:     M[0][i - 1] = minlight + lightstep * (i - 1);
149:     for (std::size_t j = 0; j < 3; ++j) { M[j + 1][i - 1] = y[j]; }
150:   }
151:
152:   return M;
153: }
154:
155:
156: // Set "nice" 2D-plot style
157: inline void SetPlotStyle() {
158:   // Set smooth color gradients
159:   const int NCont = 256;
160:
161:   const std::string style = "default";
162:
163:   if (style == "default") {
164:     const int NRGBs = 5;
165:
166:     double stops[NRGBs] = {0.00, 0.34, 0.61, 0.84, 1.00};
```

./include/Graniitti/Analysis/MROOT.h 3/4

```
167:     double red[NRGBs]   = {0.00, 0.00, 0.87, 1.00, 0.51};
168:     double green[NRGBs] = {0.00, 0.81, 1.00, 0.20, 0.00};
169:     double blue[NRGBs]  = {0.51, 1.00, 0.12, 0.00, 0.00};
170:     TColor::CreateGradientColorTable(NRGBs, stops, red, green, blue, NCont);
171:
172:     // See https://root.cern.ch/doc/master/classTColor.html
173:     // gStyle->SetPalette(53);       // kDarkBodyRadiator
174:     // gStyle->SetPalette(51);       // kDeepSea
175:     // gStyle->SetPalette(87);       // kLightTemperature
176:     // gStyle->SetPalette(105);      // kThermometer
177:     // gStyle->SetPalette(71);       // kBlueGreenYellow
178:     // gStyle->SetPalette(57);       // kBird
179:     // gStyle->SetPalette(75);       // kCherry
180:     gStyle->SetPalette(112);  // kViridis
181:
182:     // TColor::InvertPalette();      // Palette inversion
183:   } else if (style == "gray") {
184:     const int NRGBs = 5;
185:
186:     double stops[NRGBs] = {0.00, 0.34, 0.61, 0.84, 1.00};
187:     double red[NRGBs]   = {1.00, 0.84, 0.61, 0.34, 0.00};
188:     double green[NRGBs] = {1.00, 0.84, 0.61, 0.34, 0.00};
189:     double blue[NRGBs]  = {1.00, 0.84, 0.61, 0.34, 0.00};
190:     TColor::CreateGradientColorTable(NRGBs, stops, red, green, blue, NCont);
191:   } else if (style == "cubehelix") {
192:     const int                    NRGBs = 256;
193:     const std::vector<std::vector<double>> M     = CubeHelix(NRGBs, 0.5, -1.5, 1.2, 1.0);
194:
195:     double stops[NRGBs];
196:     double red[NRGBs];
197:     double green[NRGBs];
198:     double blue[NRGBs];
199:
200:     for (std::size_t i = 0; i < NRGBs; ++i) {
201:       stops[i] = M[0][i];
202:       red[i]   = M[1][i];
203:       green[i] = M[2][i];
204:       blue[i]  = M[3][i];
205:     }
206:     TColor::CreateGradientColorTable(NRGBs, stops, red, green, blue, NCont);
207:   }
208:
209:   gStyle->SetNumberContours(NCont);
210:
211:   gStyle->SetTitleOffset(1.6, "x");  // title offset from axis
212:   gStyle->SetTitleOffset(1.0, "y");  //
213:   gStyle->SetTitleSize(0.03, "x");   // title size
214:   gStyle->SetTitleSize(0.035, "y");
215:   gStyle->SetTitleSize(0.03, "z");
216:   gStyle->SetLabelOffset(0.025);
217:
218:   // Necessary with multiple plots per canvas
219:   gStyle->SetPadTopMargin(0.1);
220:   gStyle->SetPadBottomMargin(0.15);
221:
222:   gStyle->SetPadLeftMargin(0.1);
223:   gStyle->SetPadRightMargin(0.09);
224: }
225:
226: // Global Style Setup
227: inline void SetROOTStyle() {
228:   gStyle->SetOptStat(0);  // Statistics BOX OFF [0,1]
229:
230:   gStyle->SetOptFit();  // Fit parameters
231:
232:   gStyle->SetTitleSize(0.04, "t");  // Title with "t" (or anything else than xyz)
233:   gStyle->SetStatY(1.0);
234:   gStyle->SetStatX(1.0);
235:   gStyle->SetStatW(0.15);
236:   gStyle->SetStatH(0.09);
237:
238:   SetPlotStyle();
239: }
240:
241: // Before calling this, call mother TCanvas cd->()
242: inline void TransparentPad(std::shared_ptr<TPad>& pad) {
243:   pad = std::make_shared<TPad>("newpad", "a transparent pad", 0, 0, 1, 1);
244:   pad->SetFillStyle(4000);
245:   pad->Draw();
246:   pad->cd();
247: }
248:
249: // Create GRANIITTI Text
```

./include/Graniitti/Analysis/MROOT.h 4/4

```
250: inline void MadeInFinland(std::shared_ptr<TLatex>& l1, std::shared_ptr<TLatex>& l2,
251:                           double xpos = 0.935, std::vector<double> ypos = {0.03, 0.58}) {
252:   if (ypos.size() != 2) {
253:     throw std::invalid_argument("MROOT::MadeInFinland: argument ypos should be of size 2");
254:   }
255:
256:   l1 = std::make_shared<TLatex>(xpos, ypos[0], gra::aux::GetVersionTLatex().c_str());
257:   l1->SetNDC();  // Normalized coordinates
258:   l1->SetTextAngle(90);
259:   l1->Draw();
260:
261:   l2 = std::make_shared<TLatex>(xpos, ypos[1], gra::aux::GetWebTLatex().c_str());
262:   l2->SetNDC();  // Normalized coordinates
263:   l2->SetTextAngle(90);
264:   l2->Draw();
265: }
266:
267: }  // namespace rootstyle
268: }  // namespace gra
269:
270: #endif
```

./include/Graniitti/Analysis/MAnalyzer.h 1/2

```
  1: // Fast analysis class
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: #ifndef MANALYZER_H
  7: #define MANALYZER_H
  8:
  9: #include <complex>
 10: #include <memory>
 11: #include <vector>
 12:
 13: // ROOT
 14: #include "TBranch.h"
 15: #include "TCanvas.h"
 16: #include "TColor.h"
 17: #include "TF1.h"
 18: #include "TFile.h"
 19: #include "TH1.h"
 20: #include "TH2.h"
 21: #include "TLegend.h"
 22: #include "TLorentzVector.h"
 23: #include "TProfile.h"
 24: #include "TROOT.h"
 25: #include "TRandom3.h"
 26: #include "TString.h"
 27: #include "TStyle.h"
 28: #include "TTree.h"
 29:
 30: // HepMC33
 31: #include "HepMC3/FourVector.h"
 32: #include "HepMC3/GenEvent.h"
 33: #include "HepMC3/GenParticle.h"
 34: #include "HepMC3/GenVertex.h"
 35: #include "HepMC3/Print.h"
 36: #include "HepMC3/ReaderAscii.h"
 37: #include "HepMC3/Relatives.h"
 38: #include "HepMC3/Selector.h"
 39: #include "HepMC3/WriterAscii.h"
 40:
 41: // Own
 42: #include "Graniitti/Analysis/MMultiplet.h"
 43: #include "Graniitti/MAux.h"
 44:
 45: namespace gra {
 46:
 47: namespace analyzer {
 48:
 49: // Different Lorentz frame labels
 50: const std::vector<std::string> FRAMES = {"CM", "HX", "CS", "PG", "GJ", "LAB"};
 51:
 52: }  // namespace analyzer
 53:
 54: class MAnalyzer {
 55:  public:
 56:   // Constructor, destructor
 57:   MAnalyzer(const std::string &ID);
 58:   ~MAnalyzer();
 59:
 60:   // Default daughter particle name string
 61:   std::string pstr = "daughter";
 62:
 63:   // ------------------------------------------------------------
 64:   // Forward system quantities
 65:   std::shared_ptr<TH1D> hE_Pions;
 66:   std::shared_ptr<TH1D> hE_Gamma;
 67:   std::shared_ptr<TH1D> hE_Neutron;
 68:   std::shared_ptr<TH1D> hE_GammaNeutron;
 69:
 70:   std::shared_ptr<TH1D> hXF_Pions;
 71:   std::shared_ptr<TH1D> hXF_Gamma;
 72:   std::shared_ptr<TH1D> hXF_Neutron;
 73:
 74:   std::shared_ptr<TH1D> hEta_Pions;
 75:   std::shared_ptr<TH1D> hEta_Gamma;
 76:   std::shared_ptr<TH1D> hEta_Neutron;
 77:   std::shared_ptr<TH1D> hM_NSTAR;
 78:
 79:   // ------------------------------------------------------------
 80:   // Angular observables
 81:   std::shared_ptr<TProfile> hPl[8];
 82:
 83:   static constexpr unsigned int NFR = 6;  // number of frames
```

./include/Graniitti/Analysis/MAnalyzer.h 2/2

```
 84:
 85:   // Correlations between frames
 86:   std::shared_ptr<TH2D> h2CosTheta[NFR][NFR];
 87:   std::shared_ptr<TH2D> h2Phi[NFR][NFR];
 88:   // ------------------------------------------------------------
 89:
 90:   // HepMC3 reader
 91:   double HepMC3_OracleFill(const std::string inputfile, unsigned int multiplicity, int finalPDG,
 92:                            unsigned int                                                MAXEVENTS,
 93:                            std::map<std::string, std::shared_ptr<h1Multiplet>> &    h1,
 94:                            std::map<std::string, std::shared_ptr<h2Multiplet>> &    h2,
 95:                            std::map<std::string, std::shared_ptr<hProfMultiplet>> &hP,
 96:                            unsigned int                                                SID);
 97:
 98:   // Plot out all local histograms
 99:   void PlotAll(const std::string &titlestr);
100:
101:   double cross_section = 0;
102:
103:   double CheckEnergyMomentum(HepMC3::GenEvent &evt) const;
104:   void   FrameObservables(double W, HepMC3::GenEvent &evt, const M4Vec &p_beam_plus,
105:                           const M4Vec &p_beam_minus, const M4Vec &p_final_plus,
106:                           const M4Vec &p_final_minus, const std::vector<M4Vec> &pip,
107:                           const std::vector<M4Vec> &pim);
108:   void   NStarObservables(double W, HepMC3::GenEvent &evt);
109:
110:  private:
111:   double sqrts = 0.0;
112:
113:   // Name of the HepMC33 input
114:   std::string inputfile;
115:
116:   // Proton excitation is turned on
117:   bool N_STAR_ON = false;
118: };
119:
120: }  // namespace gra
121:
122: #endif
```

./include/Graniitti/Analysis/MMultiplet.h 1/2

```
  1: // Multiplet of ROOT histograms
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: #ifndef MULTIPLETCLASS_H
  7: #define MULTIPLETCLASS_H
  8:
  9: // C++
 10: #include <string>
 11:
 12: // ROOT
 13: #include "TCanvas.h"
 14: #include "TH1.h"
 15: #include "TH2.h"
 16: #include "TProfile.h"
 17:
 18: // Own
 19: #include "Graniitti/MAux.h"
 20:
 21: namespace gra {
 22: // Histogram boundaries
 23: class h1Bound {
 24:  public:
 25:   h1Bound(unsigned int _N, double _min, double _max) {
 26:     N   = _N;
 27:     min = _min;
 28:     max = _max;
 29:   }
 30:   ~h1Bound() {}
 31:
 32:   unsigned int N   = 0;
 33:   double       min = 0.0;
 34:   double       max = 0.0;
 35: };
 36:
 37: class h1Multiplet {
 38:  public:
 39:   h1Multiplet(const std::string &name, const std::string &labeltext, int N, double minval,
 40:               double maxval, const std::vector<std::string> &legendtext);
 41:   ~h1Multiplet() {
 42:     for (const auto &i : gra::aux::indices(h)) { delete h[i]; }
 43:   }
 44:   void MultiFill(const std::vector<double> &x) {
 45:     for (const auto &i : gra::aux::indices(h)) { h[i]->Fill(x[i]); }
 46:   }
 47:   void MultiFill(const std::vector<double> &x, const std::vector<double> &weight) {
 48:     for (const auto &i : gra::aux::indices(h)) { h[i]->Fill(x[i], weight[i]); }
 49:   }
 50:   void NormalizeAll(const std::vector<double> &cross_section,
 51:                     const std::vector<double> &multiplier);
 52:
 53:   // Plot and save 1D-histogram Multiplet
 54:   std::vector<double> SaveFig(const std::string &fullpath, bool RATIOPLOT = true) const;
 55:
 56:   int         N_;
 57:   double      minval_;
 58:   double      maxval_;
 59:   std::string legendposition_;
 60:   std::string name_;
 61:
 62:   // Histogram pointers here
 63:   std::vector<TH1D *>      h;
 64:   std::vector<std::string> legendtext_;
 65: };
 66:
 67: class h2Multiplet {
 68:  public:
 69:   h2Multiplet(const std::string &name, const std::string &labeltext, int N1, double minval1,
 70:               double maxval1, int N2, double minval2, double maxval2,
 71:               const std::vector<std::string> &legendtext);
 72:
 73:   ~h2Multiplet() {
 74:     for (const auto &i : gra::aux::indices(h)) { delete h[i]; }
 75:   }
 76:   void MultiFill(const std::vector<std::vector<double>> &x) {
 77:     for (const auto &i : gra::aux::indices(h)) { h[i]->Fill(x[i][0], x[i][1]); }
 78:   }
 79:   void MultiFill(const std::vector<std::vector<double>> &x, const std::vector<double> &weight) {
 80:     for (const auto &i : gra::aux::indices(h)) { h[i]->Fill(x[i][0], x[i][1], weight[i]); }
 81:   }
 82:   void NormalizeAll(const std::vector<double> &cross_section,
 83:                     const std::vector<double> &multiplier);
```

./include/Graniitti/Analysis/MMultiplet.h 2/2

```
   84:
   85:   // Plot and save histogram Multiplet
   86:   double SaveFig(const std::string &fullpath, bool RATIOPLOT = true) const;
   87:
   88:   int    N1_;
   89:   double minval1_;
   90:   double maxval1_;
   91:
   92:   int    N2_;
   93:   double minval2_;
   94:   double maxval2_;
   95:
   96:   std::string name_;
   97:
   98:   // Histogram pointers here
   99:   std::vector<TH2D *>       h;
  100:   std::vector<std::string> legendtext_;
  101: };
  102:
  103: class hProfMultiplet {
  104:  public:
  105:   hProfMultiplet(const std::string &name, const std::string &labeltext, int N, double minval1,
  106:                  double maxval1, double minval2, double maxval2,
  107:                  const std::vector<std::string> &legendtext);
  108:
  109:   ~hProfMultiplet() {
  110:     for (const auto &i : gra::aux::indices(h)) { delete h[i]; }
  111:   }
  112:   void MultiFill(const std::vector<std::vector<double>> &x) {
  113:     for (const auto &i : gra::aux::indices(h)) { h[i]->Fill(x[i][0], x[i][1]); }
  114:   }
  115:   void MultiFill(const std::vector<std::vector<double>> &x, const std::vector<double> &weight) {
  116:     for (const auto &i : gra::aux::indices(h)) { h[i]->Fill(x[i][0], x[i][1], weight[i]); }
  117:   }
  118:
  119:   // Plot and save histogram Multiplet
  120:   double SaveFig(const std::string &fullpath, bool RATIOPLOT = true) const;
  121:
  122:   int    N_;
  123:   double minval1_;
  124:   double maxval1_;
  125:
  126:   double minval2_;
  127:   double maxval2_;
  128:
  129:   std::string name_;
  130:   std::string legendposition_;
  131:
  132:   // Histogram pointers here
  133:   std::vector<TProfile *>  h;
  134:   std::vector<std::string> legendtext_;
  135: };
  136:
  137: }  // namespace gra
  138:
  139: #endif
```

./include/Graniitti/Analysis/MHarmonic.h 2/2

```
   84:                       const std::string &outputpath) const;
   85:
   86:   void PlotFigures(const std::map<gra::spherical::Meta, MTensor<gra::spherical::SH>> &tensor,
   87:                    unsigned int OBSERVABLE, const std::string &TYPESTRING, int barcolor,
   88:                    const std::string &outputpath) const;
   89:
   90:   void PlotFigures2D(const std::map<gra::spherical::Meta, MTensor<gra::spherical::SH>> &tensor,
   91:                      const std::vector<int> &OBSERVABLE2, const std::string &TYPESTRING,
   92:                      int barcolor, const std::string &outputpath) const;
   93:
   94:   void Plot1DEfficiency(unsigned int OBSERVABLE, const std::string &outputpath) const;
   95:
   96:   // Parameters
   97:   HPARAM param;
   98:   int    NCOEF;
   99:
  100:  private:
  101:   MMatrix<double>                    Y_lm;         // Calculate once for speed
  102:   std::vector<gra::spherical::Omega> DATA_events;  // Input data
  103:
  104:   // ================================================================
  105:   // Needed by MINUIT fit/cost function
  106:
  107:   // Currently active
  108:   std::vector<std::size_t> DATA_ind;
  109:   std::vector<std::size_t> activecell;  // Hypercell indices
  110:   std::vector<double>      t_lm;        // Fitted moments
  111:   std::vector<double>      t_lm_error;  // Their errors
  112:
  113:   // Error and covariance matrices
  114:   MMatrix<double> errmat;
  115:   MMatrix<double> covmat;
  116:
  117:   // ================================================================
  118:
  119:   // Active moments bookkeeping here
  120:   std::vector<bool> ACTIVE;
  121:   unsigned int      ACTIVENDF = 0;
  122:
  123:   // Detector expansion tensor data
  124:   MTensor<gra::spherical::SH_DET> det_DET;
  125:   MTensor<gra::spherical::SH_DET> fid_DET;
  126:   MTensor<gra::spherical::SH_DET> fia_DET;
  127:
  128:   // Expanded data in a map for different data sources
  129:   std::map<gra::spherical::Meta, MTensor<gra::spherical::SH>> det;
  130:   std::map<gra::spherical::Meta, MTensor<gra::spherical::SH>> fid;
  131:   std::map<gra::spherical::Meta, MTensor<gra::spherical::SH>> fia;
  132:
  133:   // Discretization
  134:   std::vector<std::vector<EDGE>> grid;
  135:
  136:   // Plotting
  137:   const std::vector<int>         colors  = {46, 38, 43, 30, 25, 14, 9, 19, 29, 39, 49};
  138:   const std::vector<std::string> xlabels = {"M (GeV)", "P_{T} (GeV)", "Y"};
  139: };
  140:
  141: }  // namespace gra
  142:
  143: #endif
```

./include/Graniitti/Analysis/MHarmonic.h 1/2

```
    1: // Spherical harmonic expansion and analysis class
    2: //
    3: //
    4: // (c) 2017-2020 Mikael Mieskolainen
    5: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
    6:
    7: #ifndef MHARMONIC_H
    8: #define MHARMONIC_H
    9:
   10: // C++
   11: #include <complex>
   12: #include <random>
   13: #include <vector>
   14:
   15: // Own
   16: #include "Graniitti/MAux.h"
   17: #include "Graniitti/MDimArray.h"
   18: #include "Graniitti/MMatrix.h"
   19: #include "Graniitti/MRandom.h"
   20: #include "Graniitti/MSpherical.h"
   21: #include "Graniitti/MTensor.h"
   22:
   23: namespace gra {
   24: class MHarmonic {
   25:  public:
   26:   // Bin edges
   27:   struct EDGE {
   28:     double min = 0;
   29:     double max = 0;
   30:
   31:     double center() const { return (max + min) / 2.0; }
   32:   };
   33:
   34:   // Parameters
   35:   struct HPARAM {
   36:     std::vector<double> M  = {0, 0, 0};
   37:     std::vector<double> Y  = {0, 0, 0};
   38:     std::vector<double> PT = {0, 0, 0};
   39:
   40:     int LMAX = 2;  // Maximum spherical harmonic truncation degree (non-negative
   41:     // integer)
   42:     bool REMOVEODD = true;  // Fix odd moments to zero (due to lacking spesific spin
   43:     // states, for example)
   44:     bool REMOVENEGATIVEM = true;  // Fix negative m to zero (due to parity
   45:     // conservation, for example)
   46:     bool   EML    = false;  // Use Extended Maximum Likelihood fit
   47:     double SVDREG = 0.001;  // SVD regularization strength in algebraic inverse (put 0
   48:     // for no regularization)
   49:     double L1REG = 0.001;  // L1-norm regularization in EML fit (put 0 for no
   50:     // regularization)
   51:
   52:     void Print() {
   53:       std::cout << "HARMONIC EXPANSION PARAMETERS:" << std::endl << std::endl;
   54:
   55:       std::cout << "LMAX             = " << LMAX << std::endl;
   56:       std::cout << "REMOVEODD        = " << (REMOVEODD ? "true" : "false") << std::endl;
   57:       std::cout << "REMOVENEGATIVEM = " << (REMOVENEGATIVEM ? "true" : "false") << std::endl;
   58:       std::cout << "EML              = " << (EML ? "true" : "false") << std::endl;
   59:       std::cout << "SVDREG           = " << SVDREG << std::endl;
   60:       std::cout << "L1REG            = " << L1REG << std::endl;
   61:     }
   62:   };
   63:
   64:   // Constructor, destructor
   65:   MHarmonic();
   66:   ~MHarmonic();
   67:
   68:   void Init(const HPARAM &hp);
   69:
   70:   void HyperLoop(void (*fitfunc)(int &, double *, double &, double *, int),
   71:                  const std::vector<gra::spherical::Omega> &MC,
   72:                  const std::vector<gra::spherical::Data> &DATA, const HPARAM &hp);
   73:
   74:   void MomentFit(const gra::spherical::Meta &META, const std::vector<std::size_t> &cell,
   75:                  void (*fitfunc)(int &, double *, double &, double *, int));
   76:
   77:   double PrintOutHyperCell(const gra::spherical::Meta &META, const std::vector<std::size_t> &cell);
   78:   void   logLfunc(int &npar, double *gin, double &f, double *par, int iflag) const;
   79:   bool   PrintLoop(const std::string &output) const;
   80:   void   PlotAll(const std::string &outputpath) const;
   81:
   82:   void Plot2DExpansion(const std::map<gra::spherical::Meta, MTensor<gra::spherical::SH>> &tensor,
   83:                        unsigned int OBSERVABLE, const std::string &TYPESTRING, int barcolor,
```

./include/Graniitti/MMatOper.h 1/4

```
    1: // Simple matrix/vector operations not included in MMatrix. [HEADER ONLY file]
    2: //
    3: // (c) 2017-2020 Mikael Mieskolainen
    4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
    5:
    6: #ifndef MMATOPER_H
    7: #define MMATOPER_H
    8:
    9: // C++
   10: #include <complex>
   11: #include <iostream>
   12: #include <vector>
   13:
   14: // Own
   15: #include "Graniitti/MMath.h"
   16: #include "Graniitti/MMatrix.h"
   17:
   18: namespace gra {
   19: namespace matoper {
   20:
   21: // MATLAB style meshgrid
   22: //
   23: // X is a matrix with each row    == x
   24: // Y is a matrix with each column == y
   25: template <typename T>
   26: inline void MeshGrid(const std::vector<T> &x, const std::vector<T> &y, MMatrix<T> &X,
   27:                      MMatrix<T> &Y) {
   28:   X = MMatrix<T>(y.size(), x.size());
   29:   Y = MMatrix<T>(y.size(), x.size());
   30:
   31:   for (std::size_t i = 0; i < X.size_row(); ++i) {
   32:     for (std::size_t j = 0; j < X.size_col(); ++j) { X[i][j] = x[j]; }
   33:   }
   34:   for (std::size_t i = 0; i < Y.size_row(); ++i) {
   35:     for (std::size_t j = 0; j < Y.size_col(); ++j) { Y[i][j] = y[i]; }
   36:   }
   37: }
   38:
   39: // diag(x) * A * diag(y) product
   40: template <typename T>
   41: inline MMatrix<T> diagAdiag(const std::vector<T> &x, const MMatrix<T> &A, const std::vector<T> &y) {
   42:   // Over (i,j) of A
   43:   const std::size_t n = A.size_row();
   44:   const std::size_t m = A.size_col();
   45:
   46:   MMatrix<T> C(n, m);
   47:   for (std::size_t i = 0; i < n; ++i) {
   48:     for (std::size_t j = 0; j < m; ++j) { C[i][j] = x[i] * A[i][j] * y[j]; }
   49:   }
   50:   return C;
   51: }
   52:
   53: // Return diagonal matrix constructed from a vector
   54: template <typename T>
   55: inline MMatrix<T> Diag(const std::vector<T> &x) {
   56:   MMatrix<T> A(x.size(), x.size(), 0.0);
   57:   for (std::size_t i = 0; i < x.size(); ++i) { A[i][i] = x[i]; }
   58:   return A;
   59: }
   60:
   61: // Return L2-unit vector (|x| = 1)
   62: template <typename T>
   63: inline std::vector<T> Unit(const std::vector<T> &x) {
   64:   double norm = 0.0;
   65:   for (std::size_t k = 0; k < x.size(); ++k) { norm += gra::math::abs2(x[k]); }
   66:   norm = gra::math::msqrt(norm);  // L2-norm
   67:
   68:   std::vector<T> y = x;
   69:   for (std::size_t k = 0; k < y.size(); ++k) {
   70:     y[k] /= norm;  // normalize
   71:   }
   72:   return y;
   73: }
   74:
   75: // Return unit sum normalized (\sum_i x_i = 1)
   76: template <typename T>
   77: inline std::vector<T> Normalized(const std::vector<T> &x) {
   78:   double sum = 0.0;
   79:   for (std::size_t k = 0; k < x.size(); ++k) { sum += x[k]; }
   80:   std::vector<T> y = x;
   81:   if (sum > 0) {
   82:     for (std::size_t k = 0; k < y.size(); ++k) {
   83:       y[k] /= sum;  // normalize
```

```
  84:     }
  85:   }
  86:   return y;
  87: }
  88:
  89: // Multiply vector by scalar
  90: template <typename T, typename T2>
  91: inline void ScaleVector(std::vector<T> &A, T2 scale) {
  92:   for (std::size_t i = 0; i < A.size(); ++i) { A[i] *= scale; }
  93: }
  94:
  95: // Add a constant matrix: A + (full ones) * c
  96: template <typename T, typename T2>
  97: inline void AddConstMat(MMatrix<T> &A, T2 c) {
  98:   for (std::size_t i = 0; i < A.size_row(); ++i) {
  99:     for (std::size_t j = 0; j < A.size_col(); ++j) { A[i][j] += c; }
 100:   }
 101: }
 102:
 103: // Vector-Matrix-Vector Multiplication
 104: // where a is (1 x m)
 105: //       B is (m x n)
 106: //       c is (n x 1)
 107: template <typename T>
 108: inline T VecMatVecMultiply(const std::vector<T> &A, const MMatrix<T> &B, const std::vector<T> &C) {
 109:   const std::size_t m = A.size();
 110:   const std::size_t n = C.size();
 111:
 112:   T value = 0.0;
 113:   for (std::size_t i = 0; i < m; ++i) {
 114:     for (std::size_t j = 0; j < n; ++j) {
 115:       value += A[i] * B[i][j] * C[j];  // notice plus
 116:     }
 117:   }
 118:   return value;
 119: }
 120:
 121: // Vector-Matrix Multiplication c = aB,
 122: // where a is (1 x m)
 123: //       B is (m x p),
 124: //       which gives c (1 x p)
 125: template <typename T>
 126: inline std::vector<T> VecMatMultiply(const std::vector<T> &A, const MMatrix<T> &B) {
 127:   const std::size_t m = A.size();
 128:   const std::size_t p = B.size_col();
 129:
 130:   std::vector<T> C(p, 0.0);  // Init with zero!
 131:
 132:   // Over [j] of C
 133:   for (std::size_t j = 0; j < p; ++j) {
 134:     for (std::size_t k = 0; k < m; ++k) {
 135:       C[j] += A[k] * B[k][j];  // notice plus
 136:     }
 137:   }
 138:
 139:   return C;
 140: }
 141:
 142: // Vector-Vector multiplication (strictly not a dot product, which would have
 143: // complex conjugation for the other)
 144: template <typename T>
 145: inline T VecVecMultiply(const std::vector<T> &A, const std::vector<T> &B) {
 146:   T out = 0.0;
 147:   for (std::size_t i = 0; i < A.size(); ++i) {
 148:     out += A[i] * B[i];  // notice plus
 149:   }
 150:   return out;
 151: }
 152:
 153: // Vector Conjugate Transpose (\dagger)
 154: // X is (n x 1)
 155: // Y is (1 x n)
 156: template <typename T>
 157: inline std::vector<T> VecDagger(const std::vector<T> &X) {
 158:   const unsigned int n = X.size();
 159:   std::vector<T>     Y(n, 0.0);
 160:
 161:   // Conjugate element by element
 162:   for (std::size_t i = 0; i < n; ++i) { Y[i] = std::conj(X[i]); }
 163:   return Y;
 164: }
 165:
 166: // Cross product for two 3-vectors
```



```
 167: template <typename T>
 168: inline std::vector<T> Cross(const std::vector<T> &a, const std::vector<T> &b) {
 169:   std::vector<T> z = {a[1] * b[2] - a[2] * b[1], a[2] * b[0] - a[0] * b[2],
 170:                       a[0] * b[1] - a[1] * b[0]};
 171:   return z;
 172: }
 173:
 174: // Negate vector
 175: template <typename T>
 176: inline std::vector<T> Negat(const std::vector<T> &a) {
 177:   std::vector<T> z(a.size(), 0.0);
 178:   for (std::size_t k = 0; k < a.size(); ++k) { z[k] = -a[k]; }
 179:   return z;
 180: }
 181:
 182: // Subtract two vectors
 183: template <typename T>
 184: inline std::vector<T> Minus(const std::vector<T> &a, const std::vector<T> &b) {
 185:   std::vector<T> z(a.size(), 0.0);
 186:   for (std::size_t k = 0; k < a.size(); ++k) { z[k] = a[k] - b[k]; }
 187:   return z;
 188: }
 189:
 190: // Add two vectors
 191: template <typename T>
 192: inline std::vector<T> Plus(const std::vector<T> &a, const std::vector<T> &b) {
 193:   std::vector<T> z(a.size(), 0.0);
 194:   for (std::size_t k = 0; k < a.size(); ++k) { z[k] = a[k] + b[k]; }
 195:   return z;
 196: }
 197:
 198: // Outer (tensor) product of two vectors: u (x) v = u v^T
 199: template <typename T>
 200: inline MMatrix<T> OuterProd(const std::vector<T> &u, const std::vector<T> &v) {
 201:   const unsigned int m = u.size();
 202:   const unsigned int n = v.size();
 203:   MMatrix<T>         out(m, n);
 204:
 205:   for (std::size_t i = 0; i < m; ++i) {
 206:     for (std::size_t j = 0; j < n; ++j) { out[i][j] = u[i] * v[j]; }
 207:   }
 208:   return out;
 209: }
 210:
 211: // Kronecker tensor product of two matrices: A(x)B = C
 212: //
 213: // Dyadic product, e.g. 2 x R^3 vectors span 6-D space
 214: // cf. full R^3x3 matrix has 9 components (like product state vs entanglement)
 215: //
 216: template <typename T>
 217: inline MMatrix<T> TensorProd(const MMatrix<T> &A, const MMatrix<T> &B) {
 218:   const unsigned int rowA = A.size_row();
 219:   const unsigned int rowB = B.size_row();
 220:   const unsigned int colA = A.size_col();
 221:   const unsigned int colB = B.size_col();
 222:
 223:   MMatrix<T> C(rowA * rowB, colA * colB);
 224:
 225:   for (std::size_t i = 0; i < rowA; ++i) {
 226:     for (std::size_t j = 0; j < colA; ++j) {
 227:       for (std::size_t k = 0; k < rowB; ++k) {
 228:         for (std::size_t l = 0; l < colB; ++l) {
 229:           C[i * rowB + k][j * colB + l] = A[i][j] * B[k][l];
 230:         }
 231:       }
 232:     }
 233:   }
 234:   return C;
 235: }
 236:
 237: // Print out vector elements
 238: template <typename T>
 239: inline void PrintVector(const std::vector<T> &A, const std::string name = "") {
 240:   // Print elements
 241:   std::cout << name << " : ";
 242:   for (std::size_t i = 0; i < A.size(); ++i) {
 243:     printf("%6.3f+i%6.3f | ", std::real(A[i]), std::imag(A[i]));
 244:   }
 245:   std::cout << std::endl;
 246: }
 247:
 248: // Print out matrix elements
 249: template <typename T>
```



```
 250: inline void PrintMatrix(const MMatrix<T> &A, const std::string name = "") {
 251:   // Print elements
 252:   std::cout << name << " : ";
 253:   for (std::size_t i = 0; i < A.size_row(); ++i) {
 254:     for (std::size_t j = 0; j < A.size_col(); ++j) {
 255:       const std::string delim = (j < A.size_col() - 1) ? ", " : "";
 256:       printf("%6.3f+i%6.3f%s ", std::real(A[i][j]), std::imag(A[i][j]), delim.c_str());
 257:     }
 258:     std::cout << std::endl;
 259:   }
 260:   std::cout << std::endl;
 261: }
 262:
 263: // Print out matrix elements as two separate matrix
 264: template <typename T>
 265: inline void PrintMatrixSeparate(const MMatrix<T> &A) {
 266:   // Print real part
 267:   std::cout << "Re:" << std::endl;
 268:   for (std::size_t i = 0; i < A.size_row(); ++i) {
 269:     for (std::size_t j = 0; j < A.size_col(); ++j) {
 270:       const std::string delim = (j < A.size_col() - 1) ? ", " : "";
 271:       printf("%6.3f%s ", std::real(A[i][j]), delim.c_str());
 272:     }
 273:     std::cout << std::endl;
 274:   }
 275:   std::cout << std::endl;
 276:
 277:   // Print imaginary part
 278:   std::cout << "Im:" << std::endl;
 279:   for (std::size_t i = 0; i < A.size_row(); ++i) {
 280:     for (std::size_t j = 0; j < A.size_col(); ++j) {
 281:       const std::string delim = (j < A.size_col() - 1) ? ", " : "";
 282:       printf("%6.3f%s ", std::imag(A[i][j]), delim.c_str());
 283:     }
 284:     std::cout << std::endl;
 285:   }
 286:   std::cout << std::endl;
 287: }
 288:
 289: }  // namespace matoper
 290: }  // namespace gra
 291:
 292: #endif
```



```
   1: // Functional methods for Spherical Harmonic Expansions
   2: //
   3: // (c) 2017-2020 Mikael Mieskolainen
   4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
   5:
   6: #ifndef MSPHERICAL_H
   7: #define MSPHERICAL_H
   8:
   9: // C++
  10: #include <complex>
  11: #include <iostream>
  12: #include <vector>
  13:
  14: // Own
  15: #include "Graniitti/MMath.h"
  16: #include "Graniitti/MMatrix.h"
  17:
  18: namespace gra {
  19: namespace spherical {
  20: // Metadata
  21: struct Meta {
  22:   std::string NAME;             // Input name ID
  23:   std::string LEGEND;           // Legend string
  24:   std::string YAXIS;            // Yaxis string
  25:   std::string MODE;             // MC or DATA
  26:   bool        FASTSIM = false;  // Fast simulation on
  27:   std::string FRAME;            // Lorentz frame
  28:   double      SCALE = 1.0;      // Scale/normalization value
  29:
  30:   std::vector<std::string> TITLES;  // Phase space titles
  31:
  32:   // So we can use this in std::map<>
  33:   bool operator<(const Meta &rhs) const { return NAME < rhs.NAME; }
  34:
  35:   void Print() const {
  36:     std::cout << "NAME:     " << NAME << std::endl;
  37:     std::cout << "LEGEND:   " << LEGEND << std::endl;
  38:     std::cout << "YAXIS:    " << YAXIS << std::endl;
  39:     std::cout << "MODE:     " << MODE << std::endl;
  40:     std::cout << "FRAME:    " << FRAME << std::endl;
  41:     std::cout << "FASTSIM:  " << (FASTSIM ? "true" : "false") << std::endl;
  42:     std::cout << "SCALE:    " << SCALE << std::endl;
  43:
  44:     std::cout << std::endl;
  45:     for (std::size_t i = 0; i < TITLES.size(); ++i) {
  46:       printf("TITLES[%lu] = %s \n", i, TITLES[i].c_str());
  47:     }
  48:   }
  49: };
  50:
  51: // Microevent structure
  52: struct Omega {
  53:   // Decay daughter
  54:   // rest frame variables
  55:   double costheta = 0.0;
  56:   double phi      = 0.0;
  57:
  58:   // Invariant / system lab frame variables
  59:   double M  = 0.0;
  60:   double Pt = 0.0;
  61:   double Y  = 0.0;
  62:
  63:   bool fiducial = false;
  64:   bool selected = false;
  65: };
  66:
  67: // Container
  68: struct Data {
  69:   // Metadata
  70:   spherical::Meta META;
  71:
  72:   // Events
  73:   std::vector<spherical::Omega> EVENTS;
  74: };
  75:
  76: // Detector data for one hypercell, e.g., in (M,Pt,Y)
  77: struct SH_DET {
  78:   // Moment mixing matrix
  79:   MMatrix<double> MIXlm;
  80:
  81:   // Efficiency decomposition coefficients
  82:   std::vector<double> E_lm;
  83:   std::vector<double> E_lm_error;
```

```
 84: };
 85:
 86: // Data for one hypercell, e.g., in (M,Pt,Y)
 87: struct SH {
 88:   // Directly (algebraic) observed moments
 89:   std::vector<double> t_lm_MPP;
 90:   std::vector<double> t_lm_MPP_error;
 91:
 92:   // Extended Maximum Likelihood fitted moments
 93:   std::vector<double> t_lm_EML;
 94:   std::vector<double> t_lm_EML_error;
 95: };
 96:
 97: MMatrix<double> GetGMixing(const std::vector<Omega> &events, const std::vector<std::size_t> &ind,
 98:                            int LMAX, const std::string &mode);
 99:
100: std::pair<std::vector<double>, std::vector<double>> GetELM(const std::vector<Omega> &       MC,
101:                                                            const std::vector<std::size_t> &ind,
102:                                                            int LMAX, const std::string &mode);
103:
104: std::vector<double> SphericalMoments(const std::vector<Omega> &      input,
105:                                      const std::vector<std::size_t> &ind, int LMAX,
106:                                      const std::string &mode);
107:
108: MMatrix<double> YLM(const std::vector<Omega> &events, int LMAX);
109:
110: double HarmDotProd(const std::vector<double> &G, const std::vector<double> &x,
111:                    const std::vector<bool> &ACTIVE, int LMAX);
112:
113: void PrintOutMoments(const std::vector<double> &x, const std::vector<double> &x_error,
114:                      const std::vector<bool> &ACTIVE, int LMAX);
115:
116: std::vector<std::size_t> GetIndices(const std::vector<Omega> &events, const std::vector<double> &M,
117:                                     const std::vector<double> &Pt, const std::vector<double> &Y);
118:
119: void   TestSphericalIntegrals(int LMAX);
120: int    LinearInd(int l, int m);
121: double CalcError(double f2, double f, double N);
122: void   PrintMatrix(FILE *fp, const std::vector<std::vector<double>> &A);
123:
124: MMatrix<double> Y_real_synthesize(const std::vector<double> &c_lm, const std::vector<bool> &ACTIVE,
125:                                   int::size_t N, std::vector<double> &costheta,
126:                                   std::vector<double> &phi, bool normalized = false);
127:
128: std::vector<double> ErrorProp(const MMatrix<double> &A, const std::vector<double> &x);
129:
130: }  // namespace spherical
131: }  // namespace gra
132:
133: #endif
```



```
  1: // Mathematical functions [HEADER ONLY file]
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: #ifndef MMATH_H
  7: #define MMATH_H
  8:
  9: // C++
 10: #include <algorithm>
 11: #include <complex>
 12: #include <random>
 13: #include <stdexcept>
 14: #include <vector>
 15:
 16: // Eigen
 17: #include <Eigen/Dense>
 18:
 19: // Own
 20: #include "Graniitti/M4Vec.h"
 21: #include "Graniitti/MMatrix.h"
 22: #include "Graniitti/MTensor.h"
 23:
 24: namespace gra {
 25: namespace math {
 26: // Imaginary unit
 27: constexpr const std::complex<double> zi(0.0, 1.0);
 28:
 29: // Geometry
 30: constexpr const double PI   = 3.14159265358979323846264338327950288419716939937510582097494 4L;
 31: constexpr const double PIPI = PI * PI;
 32:
 33: // Index all combinations of numbers in vectors [recursive function]
 34: //
 35: // ------------------------------------------------------------------------
 36: // Test code:
 37: //
 38: // const std::vector<std::vector<size_t>> allind = {{0,1,2}, {0,1}, {0,1,2,3}};
 39: //
 40: // std::vector<size_t> current;
 41: // std::vector<std::vector<std::size_t>> output;
 42: // math::IndexComb(allind, 0, current, output);
 43: //
 44: // for (std::size_t i = 0; i < output.size(); ++i) {
 45: //   for (std::size_t j = 0; j < output[i].size(); ++j) {
 46: //     std::cout << output[i][j] << " ";
 47: //   }
 48: //   std::cout << std::endl;
 49: // }
 50: // ------------------------------------------------------------------------
 51: template <typename T>
 52: void IndexComb(const std::vector<std::vector<T>> &allind, std::size_t index, std::vector<T> current,
 53:                std::vector<std::vector<T>> &output) {
 54:   if (index >= allind.size()) {
 55:     output.push_back(current);
 56:     return;
 57:   }
 58:   for (std::size_t i = 0; i < allind[index].size(); ++i) {
 59:     std::vector<T> b = current;
 60:     b.push_back(allind[index][i]);
 61:     IndexComb(allind, index + 1, b, output);
 62:   }
 63: }
 64:
 65: // Moore-Penrose Pseudoinverse for Eigen
 66: template <typename T>
 67: T PseudoInverse(const T &a, double epsilon = std::numeric_limits<double>::epsilon()) {
 68:   Eigen::JacobiSVD<T> svd(a, Eigen::ComputeThinU | Eigen::ComputeThinV);
 69:
 70:   const double tolerance =
 71:       epsilon * std::max(a.cols(), a.rows()) * svd.singularValues().array().abs()(0);
 72:
 73:   return svd.matrixV() *
 74:          (svd.singularValues().array().abs() > tolerance)
 75:              .select(svd.singularValues().array().inverse(), 0)
 76:              .matrix()
 77:              .asDiagonal() *
 78:          svd.matrixU().adjoint();
 79: }
 80:
 81: // Matrix-Vector Multiplication C = AB,
 82: // where A is (n x m)
 83: //       B is (m x 1),
```



```
 84: //       which gives C (n x 1)
 85: template <typename T>
 86: inline std::vector<T> vv_vMultiply(const std::vector<std::vector<T>> &A, const std::vector<T> &B) {
 87:   const std::size_t n = A.size();
 88:   const std::size_t m = A[0].size();
 89:   std::vector<T>    C(n, 0.0);  // Init with zero!
 90:
 91:   for (std::size_t i = 0; i < n; ++i) {
 92:     for (std::size_t k = 0; k < m; ++k) {
 93:       C[i] += A[i][k] * B[k];  // notice plus
 94:     }
 95:   }
 96:   return C;
 97: }
 98:
 99: template <typename T>
100: inline std::vector<T> valarray2vector(const std::valarray<T> &x) {
101:   std::vector<T> y(x.size());
102:   for (std::size_t k = 0; k < x.size(); ++k) { y[k] = x[k]; }
103:   return y;
104: }
105: template <typename T>
106: inline std::valarray<T> vector2valarray(const std::vector<T> &x) {
107:   std::valarray<T> y(x.size());
108:   for (std::size_t k = 0; k < x.size(); ++k) { y[k] = x[k]; }
109:   return y;
110: }
111:
112: // Absolute values per vector element
113: template <typename T>
114: inline std::vector<T> vabs(const std::vector<T> &x) {
115:   std::vector<T> y(x.size());
116:   for (std::size_t k = 0; k < x.size(); ++k) { y[k] = std::abs(x[k]); }
117:   return y;
118: }
119:
120: // MATLAB style, H is std::vector, std::valarray or similar
121: // use like:
122: // std::valarray<double> a = linspace<std::valarray>(0.0, 10.0, 16);
123: //
124: template <typename T>
125: inline std::vector<T> linspace(T start, T stop, std::size_t size) {
126:   if (size == 0) { throw std::invalid_argument("linspace: argument size == 0"); }
127:   if (size == 1) { return {stop}; }
128:
129:   std::vector<T> v(size);
130:   for (std::size_t i = 0; i < size; ++i) { v[i] = start + i * (stop - start) / (size - 1); }
131:   return v;
132: }
133:
134: template <template <typename T> class container_type, class value_type>
135: inline container_type<value_type> linspace(value_type start, value_type stop, std::size_t size) {
136:   if (size == 0) { throw std::invalid_argument("linspace: argument size == 0"); }
137:   if (size == 1) { return {stop}; }
138:
139:   container_type<value_type> v(size);
140:   for (std::size_t i = 0; i < size; ++i) { v[i] = start + i * (stop - start) / (size - 1); }
141:   return v;
142: }
143: template <template <typename T> class container_type, class value_type>
144: inline container_type<value_type> arange(value_type start, value_type step, value_type stop) {
145:   std::size_t          size = (stop - start) / step;
146:   container_type<value_type> v(size);
147:   for (std::size_t i = 0; i < size; ++i) { v[i] = start + step * i; }
148:   return v;
149: }
150:
151: // Cumulative sum
152: template <typename T>
153: inline void CumSum(const std::vector<T> &x, std::vector<T> &sumvec) {
154:   const std::size_t N = x.size();
155:   sumvec.resize(N, 0.0);
156:   sumvec[0] = x[0];  // First
157:   for (std::size_t i = 1; i < N; ++i) { sumvec[i] = sumvec[i - 1] + x[i]; }
158: }
159:
160: // Explicit powers for faster evaluation (std::pow is slow)
161: template <typename T>
162: constexpr T pow2(T x) {
163:   return x * x;
164: }
165: template <typename T>
166: constexpr T pow3(T x) {
```



```
167:   return x * pow2(x);
168: }
169: template <typename T>
170: constexpr T pow4(T x) {
171:   return pow2(pow2(x));
172: }
173: template <typename T>
174: constexpr T pow5(T x) {
175:   return x * pow4(x);
176: }
177: template <typename T>
178: constexpr T pow6(T x) {
179:   return pow2(pow3(x));
180: }
181: template <typename T>
182: constexpr T pow7(T x) {
183:   return x * pow6(x);
184: }
185: template <typename T>
186: constexpr T pow8(T x) {
187:   return pow2(pow4(x));
188: }
189:
190: // Energy-Momentum conservation
191: inline bool CheckEMC(const M4Vec &diff, double epsilon = 1e-6) {
192:   if (std::abs(diff.Px()) > epsilon || std::abs(diff.Py()) > epsilon ||
193:       std::abs(diff.Pz()) > epsilon || diff.E() > epsilon) {
194:     return false;
195:   } else {
196:     return true;
197:   }
198: }
199:
200: // Factorial n! = n*(n-1)*(n-2)*...*1
201: constexpr double factorial(int n) {
202:   double x = 1;
203:   for (int i = 1; i <= n; ++i) { x *= i; }
204:   return x;
205: }
206:
207: // Complex sign function
208: constexpr double csgn(std::complex<double> z) {
209:   const double Re = std::real(z);
210:   const double Im = std::imag(z);
211:
212:   if (Re > 0.0 || (std::abs(Re) < 1e-12 && Im > 0.0)) {
213:     return 1.0;
214:   } else {
215:     return -1.0;
216:   }
217: }
218:
219: // Sign function
220: //
221: // Sign for x=0 is 0
222: //
223: constexpr double sign(double x) {
224:   const double EPS = 1e-9;
225:   if (std::abs(x) < EPS) {
226:     return 0.0;
227:   } else if (x > 0) {
228:     return 1.0;
229:   } else {
230:     return -1.0;
231:   }
232: }
233:
234: // Sgn function
235: constexpr double sign(double X, double Y) {
236:   if (Y < 0.0) {
237:     return (-std::abs(X));
238:   } else {
239:     return (std::abs(X));
240:   }
241: }
242:
243: // Sign function
244: // template <typename T>
245: // constexpr int sgn(T val) {
246: //  return (T(0) < val) - (val < T(0));
247: //}
248:
249: // Floating point rounding safe square root
```

```
250: inline double msqrt(double x) { return std::sqrt(std::max(0.0, x)); }
251:
252: // Complex amplitude squared
253: inline double abs2(const std::complex<double> &M) { return pow2(std::abs(M)); }
254:
255: // ||v||^2
256: template <typename T>
257: inline double vpow2(const std::vector<T> &v) {
258:   double sum = 0.0;
259:   for (std::size_t i = 0; i < v.size(); ++i) { sum += abs2(v[i]); }
260:   return sum;
261: }
262:
263: // ||v||
264: template <typename T>
265: inline double l2norm(const std::vector<T> &v) {
266:   double sum = vpow2(v);
267:   return msqrt(sum);
268: }
269:
270: // Template print
271: template <template <typename T> class container_type, class value_type>
272: inline void PrintArray(container_type<value_type> x, std::string name) {
273:   std::cout << "PrintArray: " << name << std::endl;
274:   for (unsigned int i = 0; i < x.size(); ++i) { std::cout << x[i]; }
275:   std::cout << std::endl << std::endl;
276: }
277:
278: // Degrees to radians
279: constexpr double Deg2Rad(double deg) { return (deg / 180.0) * gra::math::PI; }
280:
281: // Radians to degrees
282: constexpr double Rad2Deg(double rad) { return (rad * 180.0) / gra::math::PI; }
283:
284: // Binomial Coefficient C(n, k)
285: constexpr int Cbinom(int n, int k) {
286:   if (k == 0 || k == n) { return 1; }
287:
288:   // Recursive function
289:   return Cbinom(n - 1, k - 1) + Cbinom(n - 1, k);
290: }
291:
292: // N-dim epsilon tensor e_{\mu_1,\mu_2,\mu_3,...,\mu_N}:
293: //  + 1 if even permutation of arguments
294: //  - 1 if odd permutation of arguments
295: //    0 otherwise
296: inline MTensor<int> EpsTensor(std::size_t N) {
297:   // Maximum range value for each for-loop
298:   const std::size_t MAX = N;
299:
300:   // These hold for-loop index for each nested for loop
301:   std::vector<std::size_t> ind(N, 0);
302:
303:   // -------------------------------------------------------------
304:   // Permutation tensor definition
305:   auto permutation = [&]() {
306:     int value = 1;
307:     for (std::size_t i = 0; i < N; ++i) {
308:       for (std::size_t j = i + 1; j < N; ++j) {
309:         // Even permutation +1, Odd permutation -1, Otherwise 0
310:         if (ind[i] > ind[j]) {
311:           value = -value;
312:         } else if (ind[i] == ind[j]) {
313:           return 0;
314:         }
315:       }
316:     }
317:     return value;
318:   };
319:   // -------------------------------------------------------------
320:
321:   const std::vector<std::size_t> dimensions(N, MAX);
322:   MTensor<int>                   T(dimensions, int(0));
323:
324:   // Nested for-loop
325:   std::size_t index = 0;
326:   while (true) {
327:     // ---------------------------------------------------------
328:     // Evaluate the function value
329:     T(ind) = permutation();
330:     // ---------------------------------------------------------
331:
332:     ind[0]++;
```



```
333:
334:     // Carry
335:     while (ind[index] == MAX) {
336:       if (index == N - 1) { return T; }
337:
338:       ind[index++] = 0;
339:       ind[index]++;
340:     }
341:     index = 0;
342:   }
343: }
344:
345: // Composite trapezoidal integral:
346: // Has geometric (=fast) convergence for periodic functions:
347: // - a circle in complex plane
348: // - interval on real line
349: // - Hankel contour around (-inf,0]
350: //
351: // hstep = (b-a)/N is the discretization size
352: //
353: // Indexing: j = 0,1,...,N-1,N,  => length of f = N+1, N = even
354: //
355: template <typename T>
356: inline T CTrapIntegral(const std::vector<T> &f, double hstep) {
357:   if ((f.size() - 1) % 2 != 0) {  // Must be even
358:     std::string str = "FATAL ERROR: gra::math::CTrapIntegral N = " + std::to_string(f.size() - 1) +
359:                       " is not even!";
360:     throw std::invalid_argument(str);
361:   }
362:
363:   const std::size_t N      = f.size() - 1;
364:   T                 I_sum = 0.0;
365:
366:   for (std::size_t j = 1; j <= N - 1; ++j) { I_sum += f[j]; }
367:   T I = hstep * (f[0] + 2 * I_sum + f[N]) / 2;
368:
369:   return I;
370: }
371:
372: // Composite Simpson's rule integral (quadratic interpolation),
373: // - hstep=(b-a)/N the discretization size
374: //
375: // Indexing: j = 0,1,...,N-1,N,  => length of f = N+1, N = even
376: //
377: template <typename T>
378: inline T CSIntegral(const std::vector<T> &f, double hstep) {
379:   if ((f.size() - 1) % 2 != 0) {  // Must be even
380:     std::string str =
381:         "FATAL ERROR: gra::math::CSIntegral N = " + std::to_string(f.size() - 1) + " is not even!";
382:     throw std::invalid_argument(str);
383:   }
384:
385:   const std::size_t N      = f.size() - 1;
386:   T                 I_even = 0.0;
387:   T                 I_odd  = 0.0;
388:
389:   for (std::size_t j = 1; j <= N / 2 - 1; ++j) { I_even += f[2 * j]; }
390:   for (std::size_t j = 1; j <= N / 2; ++j) { I_odd += f[2 * j - 1]; }
391:   T I = hstep * (f[0] + 2.0 * I_even + 4.0 * I_odd + f[N]) / 3.0;
392:
393:   return I;
394: }
395:
396: // Composite Simpson's 3/8 rule integral (qubic interpolation),
397: // - hstep=(b-a)/3N the discretization size
398: //
399: // Indexing: j = 0,1,...,3N, => length of f = 3N+1
400: //
401: template <typename T>
402: inline T CS38Integral(const std::vector<T> &f, double hstep) {
403:   if ((f.size() - 1) % 3 != 0) {  // Must be multiple of 3
404:     std::string str = "FATAL ERROR: gra::math::CS38Integral N = " +
405:                       std::to_string((f.size() - 1) / 3) + " is not multiple of 3!";
406:     throw std::invalid_argument(str);
407:   }
408:
409:   const std::size_t N = (f.size() - 1) / 3;
410:
411:   T I_first  = 0.0;
412:   T I_second = 0.0;
413:
414:   for (std::size_t j = 1; j <= N; ++j) { I_first += (f[3 * j - 2] + f[3 * j - 1]); }
415:   for (std::size_t j = 1; j <= N - 1; ++j) { I_second += f[3 * j]; }
```



```
416:   T I = 3 * hstep * (f[0] + 3 * I_first + 2 * I_second + f[3 * N]) / 8;
417:
418:   return I;
419: }
420:
421: // -------------------------------------------------------------
422: // 2D-Simpson's 1/3 rule, see functions below
423: //
424: // f, W with dim[M+1,N+1] where M,N are even
425: // hstepM, hstepN are discretization step sizes (b-a)/M, (d-c)/N
426: //
427: // Based on Fubini's theorem <https://en.wikipedia.org/wiki/Fubini%27s_theorem>
428:
429: template <typename T>
430: inline T Simpson13Integral2D(const MMatrix<T> &f, const MMatrix<double> &W, double hstepM,
431:                              double hstepN) {
432:   if (((f.size_row() - 1) % 2 != 0) || ((f.size_col() - 1) % 2 != 0)) {  // Must be even
433:     std::string str =
434:         "FATAL ERROR: gra::math::Simpson13Integral2D M,N must be "
435:         "even!";
436:     throw std::invalid_argument(str);
437:   }
438:
439:   T I = 0.0;
440:   for (std::size_t i = 0; i < f.size_row(); ++i) {
441:     for (std::size_t j = 0; j < f.size_col(); ++j) { I += W[i][j] * f[i][j]; }
442:   }
443:   I *= (hstepM * hstepN) / 9;
444:
445:   return I;
446: }
447:
448: // -------------------------------------------------------------
449: // 2D-Simpson's 3/8 rule, see functions below
450: //
451: // f, W with dim[M+1,N+1] where M,N are multiple of 3
452: // hstepM, hstepN are discretization step sizes (b-a)/M, (d-c)/N
453: //
454: // Based on Fubini's theorem <https://en.wikipedia.org/wiki/Fubini%27s_theorem>
455:
456: template <typename T>
457: inline T Simpson38Integral2D(const MMatrix<T> &f, const MMatrix<double> &W, double hstepM,
458:                              double hstepN) {
459:   if (((f.size_row() - 1) % 3 != 0) || ((f.size_col() - 1) % 3 != 0)) {
460:     std::string str =
461:         "FATAL ERROR: gra::math::Simpson38Integral2D M,N must be "
462:         "multiple of 3!";
463:     throw std::invalid_argument(str);
464:   }
465:
466:   T I = 0.0;
467:   for (std::size_t i = 0; i < f.size_row(); ++i) {
468:     for (std::size_t j = 0; j < f.size_col(); ++j) { I += W[i][j] * f[i][j]; }
469:   }
470:   I *= 9 * (hstepM * hstepN) / 64;  // 3^2, 9^2
471:   return I;
472: }
473:
474: // Example with M = 6, N = 8:
475:
476: // 1    4    2    4    2    4    2    4    1
477: // 4   16    8   16    8   16    8   16    4
478: // 2    8    4    8    4    8    4    8    2
479: // 4   16    8   16    8   16    8   16    4
480: // 2    8    4    8    4    8    4    8    2
481: // 4   16    8   16    8   16    8   16    4
482: // 1    4    2    4    2    4    2    4    1
483:
484: // Simpson's 1/3 rule weight vector dim[N+1]
485: inline std::vector<double> Simpson13Weight(unsigned int N) {
486:   if (N % 2 != 0) {  // Must be even
487:     std::string str =
488:         "FATAL ERROR: gra::math::SimpsonWeight N = " + std::to_string(N) + " is not even!";
489:     throw std::invalid_argument(str);
490:   }
491:   std::vector<double> W(N + 1, 2.0);  // Init with 2
492:
493:   // Set 1 as the first and the last
494:   W[0] = 1.0;
495:   W[N] = 1.0;
496:
497:   for (std::size_t i = 1; i < N; i += 2) { W[i] = 4.0; }
498:   return W;
```



```
499: }
500:
501: // Simpson's 3/8 rule weight vector dim[N+1]
502: inline std::vector<double> Simpson38Weight(unsigned int N) {
503:   if (N % 3 != 0) {  // Must be multiple of 3
504:     std::string str =
505:         "FATAL ERROR: gra::math::SimpsonWeight N = " + std::to_string(N) + " is not multiple of 3!";
506:     throw std::invalid_argument(str);
507:   }
508:   std::vector<double> W(N + 1, 3.0);  // Init with 3
509:
510:   // Set 1 as the first and the last
511:   W[0] = 1.0;
512:   W[N] = 1.0;
513:
514:   for (std::size_t i = 3; i < N; i += 3) { W[i] = 2.0; }
515:   return W;
516: }
517:
518: // Create a weight matrix dim[M+1, N+1] for 2D-Simpson's 1/3 rule
519: inline MMatrix<double> Simpson13Weight2D(unsigned int M, unsigned int N) {
520:   MMatrix<double>        W(M + 1, N + 1, 0.0);
521:   const std::vector<double> U = Simpson13Weight(M);
522:   const std::vector<double> V = Simpson13Weight(N);
523:
524:   // Outer (tensor) product: W = U*V^T
525:   // printf("\n");
526:   for (std::size_t i = 0; i <= M; ++i) {
527:     for (std::size_t j = 0; j <= N; ++j) { W[i][j] = U[i] * V[j]; }
528:   }
529:   return W;
530: }
531:
532: // Create a weight matrix dim[M+1, N+1] for 2D-Simpson's 3/8 rule
533: inline MMatrix<double> Simpson38Weight2D(unsigned int M, unsigned int N) {
534:   MMatrix<double>        W(M + 1, N + 1, 0.0);
535:   const std::vector<double> U = Simpson38Weight(M);
536:   const std::vector<double> V = Simpson38Weight(N);
537:
538:   // Outer (tensor) product: W = U*V^T
539:   // printf("\n");
540:   for (std::size_t i = 0; i <= M; ++i) {
541:     for (std::size_t j = 0; j <= N; ++j) { W[i][j] = U[i] * V[j]; }
542:   }
543:   return W;
544: }
545:
546: // Construct binary reflect Gray code (BRGC)
547: // which is the Hamiltonian Path on a N-dim unit hypercube (2^N Boolean vector
548: // space)
549: // The operator ^ is Exclusive OR (XOR) and the operator >> is bit shift right
550: constexpr unsigned int Binary2Gray(unsigned int number) { return number ^ (number >> 1); }
551:
552: // Convert BRGC (binary reflect Gray code) to a binary number
553: // Gray code is obtained as XOR (^) with all more significant bits
554: constexpr unsigned int Gray2Binary(unsigned int number) {
555:   for (unsigned int mask = number >> 1; mask != 0; mask = mask >> 1) { number = number ^ mask; }
556:   return number;
557: }
558:
559: // Boolean vector to index representation
560: // [the normal order: 0 ^ 00, 1 ^ 01, 2 ^ 10, 3 ^ 11 ...]
561: inline int Vec2Ind(std::vector<bool> vec) {
562:   // Swap bit direction (comment this for the reverse ordering)
563:   std::reverse(vec.begin(), vec.end());
564:
565:   int retval = 0;
566:   int i      = 0;
567:
568:   for (std::vector<bool>::iterator it = vec.begin(); it != vec.end(); ++it, ++i) {
569:     if (*it) { retval |= 1 << i; }
570:   }
571:   return retval;
572: }
573:
574: // Index to Boolean vector representation
575: // [the normal order: 0 ^ 00, 1 ^ 01, 2 ^ 10, 3 ^ 11 ...]
576: // Input: x = index representation
577: //        d = Boolean vector space dimension
578: inline std::vector<bool> Ind2Vec(unsigned int ind, unsigned int d) {
579:   std::vector<bool> ret;
580:
581:   while (ind) {
```

```
582:     if (ind & 1)
583:       ret.push_back(1);
584:     else
585:       ret.push_back(0);
586:
587:     ind >>= 1;
588:   }
589:   reverse(ret.begin(), ret.end());
590:
591:   // Now will the d-dimensional bit vector starting from rightmost bit
592:   std::vector<bool> final(d, 0);
593:   for (std::size_t i = 0; i < ret.size(); ++i) { final[d - 1 - i] = ret[ret.size() - 1 - i]; }
594:
595:   return final;
596: }
597:
598: // OEIS.org A030109 sequence of left-right bit reversed sequence
599: // Input: dim = Boolean vector space dimension
600: inline std::vector<unsigned int> LRsequence(unsigned int dim) {
601:   // The sequence
602:   std::vector<unsigned int> seq(std::pow(2, dim), 0);
603:
604:   for (std::size_t i = 0; i < seq.size(); ++i) {
605:     // Write i in binary with dimension d
606:     std::vector<bool> vec = Ind2Vec(i, dim);
607:
608:     // Reverse the bits
609:     reverse(vec.begin(), vec.end());
610:
611:     // Turn back to index representation
612:     seq[i] = Vec2Ind(vec);
613:   }
614:
615:   return seq;
616: }
617:
618: // Construct binary matrix in normal binary order
619: inline std::vector<std::vector<int>> BinaryMatrix(unsigned int d) {
620:   std::vector<std::vector<int>> B;
621:   unsigned int                  N = std::pow(2, d);
622:
623:   // First initialize
624:   std::vector<int> rvec(d, 0);
625:   for (std::size_t i = 0; i < N; ++i) { B.push_back(rvec); }
626:
627:   // Now fill
628:   for (std::size_t i = 0; i < N; ++i) {
629:     // Get binary expansion (vector) for this i = 0...2^d-1
630:     std::vector<bool> binvec = Ind2Vec(i, d);
631:
632:     for (unsigned int j = 0; j < d; ++j) { B[i][j] = binvec[j]; }
633:   }
634:   return B;
635: }
636:
637: // Assumes big endian
638: // Example of how to use: int i = 6; PrintBits(sizeof(i), &i);
639: // >> 00000000000000000000000000000110
640: inline void PrintBits(size_t const size, void const *const ptr) {
641:   unsigned char *b = (unsigned char *)ptr;
642:   unsigned char  byte;
643:
644:   // Loop over bytes, e.g. for int32 the size is 4
645:   for (int i = size - 1; i >= 0; i--) {
646:     // Loop over bits of a byte
647:     for (int j = 7; j >= 0; j--) {
648:       byte = (b[i] >> j) & 1;
649:       printf("%u", byte);
650:     }
651:   }
652:   puts("");
653: }
654:
655:
656: // Go through all the permutations, N! number of them
657: // Algorithm from the Knuth's "Art of Computer Programming"
658: //
659: // Input example:
660: //
661: // std::vector<int> set = {1,2,3};
662: inline std::vector<std::vector<int>> Permutations(std::vector<int> &input) {
663:   std::size_t N           = input.size();
664:   int         permutations = (int)factorial(N);
```



```
665:
666:   // Here we save all permutations
667:   std::vector<std::vector<int>> outset(permutations, std::vector<int>(N));
668:   // This we modify
669:   std::vector<int> set = input;
670:
671:   int temp = 0;
672:   int j    = 0;
673:   int l    = 0;
674:
675:   // Save first
676:   outset[0] = set;
677:   int k     = 1;
678:
679:   while (true) {
680:     // Find largest j for which set[j] < set[j+1]
681:     j = -1;
682:     for (std::size_t i = 0; i < N - 1; ++i) {
683:       if (set[i] < set[i + 1]) j = i;
684:     }
685:     if (j == -1) {
686:       break;  // No such j, we are done
687:     }
688:     for (std::size_t i = j + 1; i < N; ++i) {
689:       if (set[i] > set[j]) { l = i; }
690:     }
691:     temp   = set[j];
692:     set[j] = set[l];
693:     set[l] = temp;
694:
695:     // Reverse from j+1 to the end
696:     unsigned int swaps = (N - j - 1) / 2;
697:     for (unsigned int i = 0; i < swaps; ++i) {
698:       if ((j + 1 + i) < N) {
699:         temp           = set[(j + 1) + i];
700:         set[(j + 1) + i] = set[N - 1 - i];
701:         set[N - 1 - i]   = temp;
702:       }
703:     }
704:     // Save permutation set
705:     outset[k] = set;
706:     ++k;
707:   }
708:   return outset;
709: }
710:
711: // type == 0 : Returns charge conserving amplitude permutations, such as
712: // pi+pi-pi+pi- final state
713: // For N = 2,4,6,8 this gives number of 2,16,288,9216 permutations of final
714: // state amplitudes (legs)
715: // which is OEIS sequence
716: // A055546 (absolute coefficients of Cayley-Menger determinant of order N)
717: //
718:
719: // type == 1 : Returns fully symmetrized amplitude, such as the case of yyyy (4
720: // gammas) final state
721: // This gives 2! 4! 6! (4,24,720) ... (pure factorial) number of permutations
722: //
723: inline std::vector<std::vector<int>> GetAmpPerm(int N, int type) {
724:   // We start indexing at 3 for central system particles relevant for the
725:   // amplitude
726:   // indexing: [0 not in use, 1,2 are final state protons]
727:   int offset = 3;
728:
729:   std::vector<int> set;
730:   int              k = offset;
731:   for (int i = 0; i < N; ++i) {
732:     set.push_back(k);
733:     ++k;
734:   }
735:   // Get complete permutations
736:   std::vector<std::vector<int>> outset = Permutations(set);
737:
738:   if (type == 1) {
739:     return outset;  // Return complete permutations
740:   } else {          // Return local charge conserving
741:
742:     // Create alternating charge series
743:     // 1,-1,1,-1,...
744:     std::vector<int> charge;
745:     for (int i = 0; i < N; ++i) {
746:       charge.push_back(pow(-1, i));
747:       //  printf("%d", charge[i]);
```



```
748:     }
749:     // printf("\n");
750:
751:     std::vector<std::vector<int>> valid_amplitudes;
752:
753:     for (std::size_t i = 0; i < outset.size(); ++i) {
754:       bool ok = true;
755:       for (std::size_t j = 0; j < outset[0].size() - 1; j += 2) {
756:         int a = outset[i][j];
757:         int b = outset[i][j + 1];
758:
759:         if ((charge.at(a - offset) + charge.at(b - offset)) != 0) ok = false;
760:         // printf("%d%d", a, b);
761:       }
762:       if (ok) { valid_amplitudes.push_back(outset[i]); }
763:       //  printf(" ok = %d", ok);
764:       //  printf("\n");
765:     }
766:     // printf("Total = %d, Charge conserving = %d \n",
767:     // outset.size(), valid_amplitudes.size());
768:     return valid_amplitudes;
769:   }
770: }
771:
772: // Ordinary Legendre polynomials P_l(x) to cross check algorithmic
773: // implementation
774: // x usually cos(theta)
775: inline double LegendreP1(unsigned int l, double x) {
776:   if (l == 0) {
777:     return 1;
778:   } else if (l == 1) {
779:     return x;
780:   } else if (l == 2) {
781:     return 0.5 * (3 * x * x - 1);
782:   } else if (l == 3) {
783:     return 0.5 * (5 * x * x * x - 3 * x);
784:   } else if (l == 4) {
785:     return (35 * x * x * x * x - 30 * x * x + 3) / 8;
786:   } else if (l == 5) {
787:     return (63 * x * x * x * x * x - 70 * x * x * x + 15 * x) / 8;
788:   } else if (l == 6) {
789:     return (231 * x * x * x * x * x * x - 315 * x * x * x * x + 105 * x * x - 5) / 16;
790:   } else if (l == 7) {
791:     return (429 * x * x * x * x * x * x * x - 693 * x * x * x * x * x + 315 * x * x * x - 35 * x) /
792:            16;
793:   } else if (l == 8) {
794:     return (6435 * x * x * x * x * x * x * x * x - 12012 * x * x * x * x * x * x +
795:             6930 * x * x * x * x - 1260 * x * x + 35) /
796:            128;
797:   } else {
798:     throw std::invalid_argument("LegendreP1: Not valid l = " + std::to_string(l));
799:   }
800: }
801:
802: // Associated Legendre Polynomial P_l^m(x) solved numerically
803: //
804: // Algorithm from:
805: //
806: // [REFERENCE: S. Jemma, The Nitty Gritty Details of Spherical Harmonics, SIGRAPH2003]
807: constexpr double sf_legendre(int l, int m, double x) {
808:   // Associated Legendre Polynomial
809:   double pmm = 1.0;
810:   if (m > 0) {
811:     const double sx2  = std::sqrt((1.0 - x) * (1.0 + x));
812:     double       fact = 1.0;
813:     for (int i = 1; i <= m; ++i) {
814:       pmm *= (-fact) * sx2;
815:       fact += 2.0;
816:     }
817:   }
818:   if (l == m) { return pmm; }
819:
820:   double pmmp1 = x * (2.0 * m + 1.0) * pmm;
821:   if (l == m + 1) { return pmmp1; }
822:
823:   double pll = 0.0;
824:   for (int ll = m + 2; ll <= l; ++ll) {
825:     pll   = ((2.0 * ll - 1.0) * x * pmmp1 - (ll + m - 1.0) * pmm) / (ll - m);
826:     pmm   = pmmp1;
827:     pmmp1 = pll;
828:   }
829:   return pll;
830: }
```



```
831:
832: // ------------------------------------------------------------------------
833: // Associated Legendre Polynomials with Spherical Harmonics
834: //
835: //
836: // Quantum Mechanics convention
837: //  Y_l^m(theta,phi)
838: //  = (-1)^m \sqrt{\frac{2l+1}{4\pi} \frac{(l-m)!}{(l+m)!} P_{lm}(cos(theta))
839: //  e^{im\phi}
840: //
841: // Normalization:
842: //  d\Omega = sin(theta) d\phi d\theta
843: //  \int |Y_l^m|^2 d\Omega = 1
844: //
845: // Orthogonality:
846: //  \int_{\theta = 0}^\pi \int_{\phi=0}^{2\pi} Y_l^m Y_l^{m'}^* d\Omega =
847: //  \delta_{ll'} \delta_{mm'}
848: //
849: //
850: // Relation:
851: //  Y_l^{m*}(theta,phi) = (-1)^m Y_l^{-m} (theta,phi)
852: //
853: // When m = 0, we get ordinary Legendre polynomials
854: //
855: // Parity of spherical harmonics:
856: //        (r, \theta, \phi) -> (r, \pi - \theta, \pi + \phi)
857: //
858: //
859: // Y_l^m(-\vec{r}) = (-1)^l Y_l^m(\vec{r})
860: // Y_l^m(theta,phi) -> Y_l^m (\pi - \theta, \pi + \phi) = (-1)^l Y_l^m(\theta,
861: // \phi)
862: //
863: //
864: // Parity = (-1)^l, so parity is even or odd, depending on the quantum number l
865: //
866: // Real valued representation of complex spherical harmonics
867: //
868: // Normalization such that \int d\Omega |Y_l^m(\Omega)|^2 = 1
869: inline double NReY(const std::complex<double> &Y, int l, int m) {
870:   if (m < 0) {
871:     return sqrt(2.0) * std::pow(-1, m) * std::imag(Y);
872:   } else if (m == 0) {
873:     return std::real(Y);
874:   } else {
875:     return sqrt(2.0) * std::pow(-1, m) * std::real(Y);
876:   }
877: }
878:
879: // Real Basis Laplace Spherical Harmonics
880: //
881: // [REFERENCE: https://en.wikipedia.org/wiki/Spherical_harmonics]
882: inline double Y_real_basis(double costheta, double phi, int l, int m) {
883:   // Normalization function
884:   auto K = [](int l, int m) -> double {
885:
886:     if (m == 0) { return msqrt((2.0 * l + 1.0) / (4.0 * PI)); }  // Speed it up
887:     double temp =
888:         ((2.0 * l + 1.0) * factorial(l - m)) / static_cast<double>(4.0 * PI * factorial(l + m));
889:     return msqrt(temp);
890:   };
891:   const double sqrt2 = msqrt(2.0);
892:
893:   if (m < 0) {
894:     return std::pow(-1, m) * sqrt2 * K(l, -m) * std::sin(-m * phi) * sf_legendre(l, -m, costheta);
895:   } else if (m == 0) {
896:     return K(l, 0) * sf_legendre(l, m, costheta);
897:   } else {  // m > 0
898:     return std::pow(-1, m) * sqrt2 * K(l, m) * std::cos(m * phi) * sf_legendre(l, m, costheta);
899:   }
900: }
901:
902: // Complex Basis Laplace Spherical Harmonics
903: inline std::complex<double> Y_complex_basis(double costheta, double phi, int l, int m) {
904:   if (m < 0) {
905:     return 1.0 / msqrt(2.0) *
906:            (Y_real_basis(costheta, phi, l, std::abs(m)) -
907:             zi * Y_real_basis(costheta, phi, l, -std::abs(m)));
908:   } else if (m == 0) {
909:     return Y_real_basis(costheta, phi, l, 0);
910:   } else {  // m > 0
911:     return std::pow(-1, m) / msqrt(2.0) *
912:            (Y_real_basis(costheta, phi, l, std::abs(m)) +
913:             zi * Y_real_basis(costheta, phi, l, -std::abs(m)));
```

./include/Graniitti/MMath.h 12/13

```
914:   }
915: }
916:
917: // Manually written Spherical Harmonics to cross-check algorithmic
918: // implementations
919: inline std::complex<double> Y_complex_basis_ref(double costheta, double phi, int l, int m) {
920:   const double theta = std::acos(costheta);
921:
922:   if (l == 0 && m == 0) return 0.5 * std::sqrt(1.0 / PI);
923:
924:   // -----------------------------------------------------------------
925:   if (l == 1 && m == -1)
926:     return 0.5 * std::sqrt(3.0 / (2.0 * PI)) * std::exp(-gra::math::zi * phi) * std::sin(theta);
927:
928:   if (l == 1 && m == 0) return 0.5 * std::sqrt(3.0 / PI) * costheta;
929:
930:   if (l == 1 && m == 1)
931:     return -0.5 * std::sqrt(3.0 / (2.0 * PI)) * std::exp(gra::math::zi * phi) * std::sin(theta);
932:
933:   // -----------------------------------------------------------------
934:   if (l == 2 && m == -2)
935:     return 0.25 * std::sqrt(15.0 / (2.0 * PI)) * std::exp(-2.0 * gra::math::zi * phi) *
936:            std::pow(std::sin(theta), 2);
937:
938:   if (l == 2 && m == -1)
939:     return 0.5 * std::sqrt(15.0 / (2.0 * PI)) * std::exp(-gra::math::zi * phi) * std::sin(theta) *
940:            costheta;
941:
942:   if (l == 2 && m == 0) return 0.25 * std::sqrt(5.0 / PI) * (3.0 * std::pow(costheta, 2) - 1.0);
943:
944:   if (l == 2 && m == 1)
945:     return -0.5 * std::sqrt(15.0 / (2.0 * PI)) * std::exp(gra::math::zi * phi) * std::sin(theta) *
946:            costheta;
947:
948:   if (l == 2 && m == 2)
949:     return 0.25 * std::sqrt(15.0 / (2.0 * PI)) * std::exp(2.0 * gra::math::zi * phi) *
950:            std::pow(std::sin(theta), 2);
951:
952:   // -----------------------------------------------------------------
953:
954:   if (l == 3 && m == -3)
955:     return 1.0 / 8.0 * std::sqrt(35.0 / PI) * std::exp(-3.0 * gra::math::zi * phi) *
956:            std::pow(std::sin(theta), 3);
957:
958:   if (l == 3 && m == -2)
959:     return 0.25 * std::sqrt(105.0 / (2.0 * PI)) * std::exp(-2.0 * gra::math::zi * phi) *
960:            std::pow(std::sin(theta), 2) * costheta;
961:
962:   if (l == 3 && m == -1)
963:     return 1.0 / 8.0 * std::sqrt(21.0 / PI) * std::exp(-gra::math::zi * phi) * std::sin(theta) *
964:            (5.0 * std::pow(costheta, 2) - 1.0);
965:
966:   if (l == 3 && m == 0)
967:     return 0.25 * std::sqrt(7.0 / PI) * (5.0 * std::pow(costheta, 3) - 3.0 * costheta);
968:
969:   if (l == 3 && m == 1)
970:     return -1.0 / 8.0 * std::sqrt(21.0 / PI) * std::exp(gra::math::zi * phi) * std::sin(theta) *
971:            (5.0 * std::pow(costheta, 2) - 1.0);
972:
973:   if (l == 3 && m == 2)
974:     return 0.25 * std::sqrt(105.0 / (2.0 * PI)) * std::exp(2.0 * gra::math::zi * phi) *
975:            std::pow(std::sin(theta), 2) * costheta;
976:
977:   if (l == 3 && m == 3)
978:     return -1.0 / 8.0 * std::sqrt(35.0 / PI) * std::exp(3.0 * gra::math::zi * phi) *
979:            std::pow(std::sin(theta), 3);
980:
981:   // -----------------------------------------------------------------
982:
983:   if (l == 4 && m == -4)
984:     return 3.0 / 16.0 * std::sqrt(35.0 / (2.0 * PI)) * std::exp(-4.0 * gra::math::zi * phi) *
985:            std::pow(std::sin(theta), 4);
986:
987:   if (l == 4 && m == -3)
988:     return 3.0 / 8.0 * std::sqrt(35.0 / (PI)) * std::exp(-3.0 * gra::math::zi * phi) *
989:            std::pow(std::sin(theta), 3) * costheta;
990:
991:   if (l == 4 && m == -2)
992:     return 3.0 / 8.0 * std::sqrt(5.0 / (2.0 * PI)) * std::exp(-2.0 * gra::math::zi * phi) *
993:            std::pow(std::sin(theta), 2) * (7.0 * std::pow(costheta, 2) - 1.0);
994:
995:   if (l == 4 && m == -1)
996:     return 3.0 / 8.0 * std::sqrt(5.0 / PI) * std::exp(-gra::math::zi * phi) * std::sin(theta) *
```

./include/Graniitti/MMath.h 13/13

```
997:            (7.0 * std::pow(costheta, 3) - 3.0 * costheta);
998:
999:   if (l == 4 && m == 0)
1000:     return 3.0 / 16.0 * std::sqrt(1.0 / PI) *
1001:            (35.0 * std::pow(costheta, 4) - 30.0 * std::pow(costheta, 2) + 3.0);
1002:
1003:   if (l == 4 && m == 1)
1004:     return -3.0 / 8.0 * std::sqrt(5.0 / PI) * std::exp(gra::math::zi * phi) * std::sin(theta) *
1005:            (7.0 * std::pow(costheta, 3) - 3.0 * costheta);
1006:
1007:   if (l == 4 && m == 2)
1008:     return 3.0 / 8.0 * std::sqrt(5.0 / (2.0 * PI)) * std::exp(2.0 * gra::math::zi * phi) *
1009:            std::pow(std::sin(theta), 2) * (7.0 * std::pow(costheta, 2) - 1.0);
1010:
1011:   if (l == 4 && m == 3)
1012:     return -3.0 / 8.0 * std::sqrt(35.0 / (PI)) * std::exp(3.0 * gra::math::zi * phi) *
1013:            std::pow(std::sin(theta), 3) * costheta;
1014:
1015:   if (l == 4 && m == 4)
1016:     return 3.0 / 16.0 * std::sqrt(35.0 / (2.0 * PI)) * std::exp(4.0 * gra::math::zi * phi) *
1017:            std::pow(std::sin(theta), 4);
1018:
1019:   throw std::invalid_argument("Y_complex_basis_ref:: Not supported l = " + std::to_string(l) +
1020:                               ", m = " + std::to_string(m));
1021: }
1022:
1023: }  // namespace math
1024: }  // namespace gra
1025:
1026: #endif
```

./include/Graniitti/MQED.h 1/1

```
1: // Running QED coupling [HEADER ONLY FILE]
2: //
3: // (c) 2017-2020 Mikael Mieskolainen
4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
5:
6:
7: #ifndef MQED_H
8: #define MQED_H
9:
10: #include <vector>
11:
12: // Own
13: #include "Graniitti/MMath.h"
14:
15:
16: namespace gra {
17: namespace qed {
18:
19: constexpr double alpha_ref = 0.0072973525693;  // reference value of alpha at scale Q
20: constexpr double Q_ref     = 0.000510998961;  // reference scale (GeV)
21: constexpr double alpha_0 =
22:     0.0072973525693;  // value of alpha at Q = 0 (fine structure constant)^(-1)
23:
24: // ---------------------------------------------------------------------
25:
26: // Get number of charged leptons at given scale
27: inline int get_N_leptons(double Q) {
28:   if (Q > 1.77686E+00) { return 3; }
29:   if (Q > 1.056583745E-01) { return 2; }
30:   return 1;
31: }
32:
33: // The coefficient of one loop beta function
34: inline double beta0_QED(int N_lf) { return -1.0 / (3.0 * math::PI) * N_lf; }
35:
36: // QED running coupling
37: inline double alpha_QED(double Q2, const std::string& order = "LL") {
38:   // Freeze the coupling at Q < Q_ref
39:   if (order == "ZERO" || Q2 < (Q_ref * Q_ref)) { return alpha_0; }
40:
41:   // Input
42:   const double Q    = math::msqrt(Q2);
43:   const int    N_lf = get_N_leptons(Q);
44:
45:   // Beta-function quantities
46:   const double beta0 = beta0_QED(N_lf);
47:   const double R     = std::log(Q2 / (Q_ref * Q_ref));
48:
49:   if (order == "LL") {  // leading log [one loop geometric series resummation]
50:     return alpha_ref / (1.0 + alpha_ref * beta0 * R);
51:   } else {
52:     throw std::invalid_argument("alpha_QED:: Unknown order parameter: " + order);
53:   }
54: }
55:
56: // QED coupling at Q = 0
57: inline double alpha_QED() { return alpha_QED(0, "ZERO"); }
58:
59: // QED: Electric charge in natural units ~ 0.3
60: inline double e_QED(double Q2) { return math::msqrt(alpha_QED(Q2) * 4.0 * math::PI); }
61: inline double e_QED() { return math::msqrt(alpha_QED() * 4.0 * math::PI); }
62:
63:
64: }  // namespace qed
65: }  // namespace gra
66:
67: #endif
```

./include/Graniitti/MTensorPomeron.h 1/4

```
1: // Tensor Pomeron amplitudes
2: //
3: // (c) 2017-2020 Mikael Mieskolainen
4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
5:
6: #ifndef MTENSORPOMERON_H
7: #define MTENSORPOMERON_H
8:
9: // C++
10: #include <complex>
11: #include <random>
12: #include <vector>
13:
14: // Tensor algebra
15: #include "FTensor.hpp"
16:
17: // Own
18: #include "Graniitti/M4Vec.h"
19: #include "Graniitti/MDirac.h"
20: #include "Graniitti/MGlobals.h"
21: #include "Graniitti/MKinematics.h"
22: #include "Graniitti/MMath.h"
23: #include "Graniitti/MTensor.h"
24:
25:
26: namespace gra {
27:
28: // Tensor Pomeron model parameters
29: struct MTensorPomeronParam {
30:   // Trajectory parameters
31:   double delta_P  = 0;
32:   double delta_O  = 0;
33:   double delta_1R = 0;
34:   double delta_2R = 0;
35:
36:   double ap_P  = 0;  // GeV^(-2)
37:   double ap_O  = 0;  // GeV^(-2)
38:   double ap_1R = 0;  // GeV^(-2)
39:   double ap_2R = 0;  // GeV^(-2)
40:
41:   double eta_O = 1;  // +- 1
42:
43:   // Scales
44:   double M_O  = 0;  // GeV
45:   double M_1R = 0;  // GeV
46:
47:   // Couplings
48:   double              gPNN   = 0;  // GeV^(-1), Pomeron-Proton-Proton
49:   double              gPpipi = 0;  // GeV^(-1), Pomeron-Pion-Pion
50:   double              gPKK   = 0;  // GeV^(-1), Pomeron-Kaon-Kaon
51:   std::vector<double> gPrhorho;    // [GeV^(-3), GeV^(-1)], Pomeron-Rho-Rho
52:   std::vector<double> gPphiphi;    // [GeV^(-3), GeV^(-1)], Pomeron-Phi-Phi
53:
54:   bool initialized = false;
55:
56:   // Read parameters from file
57:   void ReadParameters(const std::string &modelfile) {
58:     using json            = nlohmann::json;
59:     const std::string data = gra::aux::GetInputData(modelfile);
60:     json              j;
61:
62:     try {
63:       j = json::parse(data);
64:
65:       // JSON block identifier
66:       const std::string XID = "PARAM_TENSORPOM";
67:
68:       delta_P  = j.at(XID).at("delta_P");
69:       delta_O  = j.at(XID).at("delta_O");
70:       delta_1R = j.at(XID).at("delta_1R");
71:       delta_2R = j.at(XID).at("delta_2R");
72:
73:       ap_P  = j.at(XID).at("ap_P");
74:       ap_O  = j.at(XID).at("ap_O");
75:       ap_1R = j.at(XID).at("ap_1R");
76:       ap_2R = j.at(XID).at("ap_2R");
77:
78:       eta_O = j.at(XID).at("eta_O");
79:       M_O   = j.at(XID).at("M_O");
80:       M_1R  = j.at(XID).at("M_1R");
81:
82:       gPNN   = j.at(XID).at("gPNN");
83:       gPpipi = j.at(XID).at("gPpipi");
```

./include/Graniitti/MTensorPomeron.h 2/4

```
84:       gPKK   = j.at(XID).at("gPKK");
85:
86:       std::vector<double> a = j.at(XID).at("gPrhorho");
87:       gPrhorho             = a;
88:
89:       std::vector<double> b = j.at(XID).at("gPphiphi");
90:       gPphiphi             = b;
91:
92:       initialized = true;
93:     } catch (...) {
94:       std::string str = "MTensorPomeronParam::ReadParameters: Error parsing " + modelfile +
95:                         " (Check for extra/missing commas)";
96:       throw std::invalid_argument(str);
97:     }
98:   }
99: };
100:
101:
102: // Matrix element dimension: " GeV^" << -(2*external_legs - 8)
103: class MTensorPomeron : public MDirac {
104:  public:
105:   MTensorPomeron(gra::LORENTZSCALAR &lts, const std::string &modelfile);
106:   ~MTensorPomeron() {}
107:
108:   // Decay coupling (static so we can call it independently)
109:   static double GDecay(int J, double M, double Gamma, double mf, double BR);
110:
111:   // Amplitude squared
112:   double ME3(gra::LORENTZSCALAR &lts) const;
113:   double ME4(gra::LORENTZSCALAR &lts) const;
114:   double ME6(gra::LORENTZSCALAR &lts) const;
115:
116:   // Scalar, Pseudoscalar, Tensor coupling structures
117:   FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> iG_PPS_0() const;
118:   FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> iG_PPS_1(const M4Vec &q1, const M4Vec &q2,
119:                                                               double g_PPS) const;
120:   FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> iG_PPPS_0(const M4Vec &q1, const M4Vec &q2,
121:                                                                double g_PPPS) const;
122:   FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> iG_PPPS_1(const M4Vec &q1, const M4Vec &q2,
123:                                                                double g_PPPS) const;
124:   MTensor<std::complex<double>>                      iG_PPT_00() const;
125:   MTensor<std::complex<double>> iG_PPT_12(const M4Vec &q1, const M4Vec &q2, double g_PPT,
126:                                           int mode) const;
127:   MTensor<std::complex<double>> iG_PPT_03(const M4Vec &q1, const M4Vec &q2, double g_PPT) const;
128:   MTensor<std::complex<double>> iG_PPT_04(const M4Vec &q1, const M4Vec &q2, double g_PPT) const;
129:   MTensor<std::complex<double>> iG_PPT_05(const M4Vec &q1, const M4Vec &q2, double g_PPT) const;
130:   MTensor<std::complex<double>> iG_PPT_06(const M4Vec &q1, const M4Vec &q2, double g_PPT) const;
131:
132:   // Vertex functions
133:   FTensor::Tensor2<std::complex<double>, 4, 4> iG_vv2psps(const std::vector<M4Vec> &p,
134:                                                           int                       PDG) const;
135:
136:   FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> iG_PPS_total(
137:       const M4Vec &q1, const M4Vec &q2, double M0, const std::string &mode,
138:       const std::vector<double> &g_PPS) const;
139:   MTensor<std::complex<double>> iG_PPT_total(const M4Vec &q1, const M4Vec &q2, double M0,
140:                                              const std::vector<double> &g_PPT) const;
141:
142:   FTensor::Tensor2<std::complex<double>, 4, 4> iG_PppHE(const M4Vec &prime, const M4Vec p) const;
143:
144:   FTensor::Tensor1<std::complex<double>, 4> iG_yee(
145:       const M4Vec &prime, const M4Vec &p, const std::vector<std::complex<double>> &ubar,
146:       const std::vector<std::complex<double>> &u) const;
147:
148:   FTensor::Tensor2<std::complex<double>, 4, 4> iG_yeebary(
149:       const std::vector<std::complex<double>> &ubar, const MMatrix<std::complex<double>> &iSF,
150:       const std::vector<std::complex<double>> &v) const;
151:
152:   FTensor::Tensor1<std::complex<double>, 4> iG_ypp(
153:       const M4Vec &prime, const M4Vec &p, const std::vector<std::complex<double>> &ubar,
154:       const std::vector<std::complex<double>> &u) const;
155:
156:   FTensor::Tensor2<std::complex<double>, 4, 4> iG_Ppp(
157:       const M4Vec &prime, const M4Vec &p, const std::vector<std::complex<double>> &ubar,
158:       const std::vector<std::complex<double>> &u) const;
159:
160:   FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> iG_PppbarP(
161:       const M4Vec &prime, const std::vector<std::complex<double>> &ubar, const M4Vec &pt,
162:       const MMatrix<std::complex<double>> &iSF, const std::vector<std::complex<double>> &v,
163:       const M4Vec &p) const;
164:
165:   FTensor::Tensor2<std::complex<double>, 4, 4>       iG_Ppsps(const M4Vec &prime, const M4Vec &p,
166:                                                               double g1) const;
```

./include/Graniitti/MTensorPomeron.h 4/4

```
250:   FTensor::Tensor4<double, 4, 4, 4, 4> R_DDUU;
251:   FTensor::Tensor4<double, 4, 4, 4, 4> R_UUDD;
252:
253:   MTensor<double> T1;
254:   MTensor<double> T2;
255:   MTensor<double> T3;
256:
257:   // Parameters
258:   MTensorPomeronParam Param;
259: };
260:
261: }  // namespace gra
262:
263: #endif
```

./include/Graniitti/MTensorPomeron.h 3/4

```
167:   FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> iG_Pvv(const M4Vec &prime, const M4Vec &p,
168:                                                              double g1, double g2) const;
169:
170:   std::complex<double> iG_f0ss(const M4Vec &p3, const M4Vec &p4, double M0, double g1) const;
171:   FTensor::Tensor2<std::complex<double>, 4, 4> iG_f0vv(const M4Vec &p3, const M4Vec &p4, double M0,
172:                                                        double g1, double g2) const;
173:
174:   FTensor::Tensor2<std::complex<double>, 4, 4> iG_psvv(const M4Vec &p3, const M4Vec &p4, double M0,
175:                                                        double g1) const;
176:   FTensor::Tensor1<std::complex<double>, 4>    iG_vpsps(const M4Vec &k1, const M4Vec &k2, double M0,
177:                                                         double g1) const;
178:
179:   FTensor::Tensor2<std::complex<double>, 4, 4>       iG_f2psps(const M4Vec &k1, const M4Vec &k2,
180:                                                                double M0, double g1) const;
181:   FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> iG_f2vv(const M4Vec &k1, const M4Vec &k2,
182:                                                              double M0, double g1, double g2) const;
183:   FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> iG_f2yy(const M4Vec &k1, const M4Vec &k2,
184:                                                              double M0, double g1, double g2) const;
185:
186:   FTensor::Tensor2<std::complex<double>, 4, 4> iG_yV(double q2, int pdg) const;
187:
188:   // Polarization sums
189:   std::vector<FTensor::Tensor2<std::complex<double>, 4, 4>> MassiveSpin1PolSum(
190:       const FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> &M, const M4Vec &p3,
191:       const M4Vec &p4) const;
192:   std::vector<FTensor::Tensor2<std::complex<double>, 4, 4>> MasslessSpin1PolSum(
193:       const FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> &M, const M4Vec &p3,
194:       const M4Vec &p4) const;
195:
196:   std::vector<std::complex<double>> MasslessSpin1PolSum(
197:       const FTensor::Tensor2<std::complex<double>, 4, 4> &M, const M4Vec &p3,
198:       const M4Vec &p4) const;
199:   std::vector<std::complex<double>> MassiveSpin1PolSum(
200:       const FTensor::Tensor2<std::complex<double>, 4, 4> &M, const M4Vec &p3,
201:       const M4Vec &p4) const;
202:
203:   // Propagators
204:   FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> iD_P(double s, double t) const;
205:   FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> iD_2R(double s, double t) const;
206:   FTensor::Tensor2<std::complex<double>, 4, 4>       iD_O(double s, double t) const;
207:   FTensor::Tensor2<std::complex<double>, 4, 4>       iD_1R(double s, double t) const;
208:
209:   FTensor::Tensor2<std::complex<double>, 4, 4> iD_V(const M4Vec &p, double M0, double s34) const;
210:   std::complex<double>                         iD_MES0(const M4Vec &p, double M0) const;
211:   std::complex<double> iD_MES(const M4Vec &p, double M0, double Gamma) const;
212:   FTensor::Tensor2<std::complex<double>, 4, 4> iD_VMES(const M4Vec &p, double M0, double Gamma,
213:                                                        bool INDEX_UP) const;
214:   FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> iD_TMES(const M4Vec &p, double M0,
215:                                                              double Gamma, bool INDEX_UP) const;
216:
217:   // Tensor functions
218:   FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> Gamma0(const M4Vec &k1, const M4Vec &k2) const;
219:   FTensor::Tensor4<std::complex<double>, 4, 4, 4, 4> Gamma2(const M4Vec &k1, const M4Vec &k2) const;
220:   void                                               CalcRTensor();
221:
222:   // Trajectories
223:   double alpha_P(double t) const;
224:   double alpha_O(double t) const;
225:   double alpha_1R(double t) const;
226:   double alpha_2R(double t) const;
227:
228:   // Form factors
229:   double F1(double t) const;
230:   double F2(double t) const;
231:
232:   double F1_(double t) const;
233:   double F2_(double t) const;
234:
235:   double GD(double t) const;
236:   double FM(double q2, double LAMBDA) const;
237:   double F(double q2, double M0, double LAMBDA) const;
238:
239:  private:
240:   // Minkowski metric tensor
241:   FTensor::Tensor2<double, 4, 4> gT;
242:
243:   // Epsilon tensors
244:   FTensor::Tensor4<double, 4, 4, 4, 4> eps_lo;
245:   FTensor::Tensor4<double, 4, 4, 4, 4> eps_hi;
246:
247:   // Aux tensors
248:   FTensor::Tensor4<double, 4, 4, 4, 4> R_DDDD;
249:   FTensor::Tensor4<double, 4, 4, 4, 4> R_DDDU;
```

D.4 C++ source files

./src/Program/gr.cc 1/4

```
// GRANIITTI - Monte Carlo event generator for high energy diffraction
// https://github.com/mieskolainen/graniitti
//
// <Main generator program>
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

// C++
#include <chrono>
#include <iostream>
#include <memory>
#include <stdexcept>

// Own
#include "Graniitti/MAux.h"
#include "Graniitti/MGraniitti.h"
#include "Graniitti/MMatrix.h"

// Libraries
#include "cxxopts.hpp"
#include "json.hpp"
#include "rang.hpp"

using namespace gra;

// Main
int main(int argc, char *argv[]) {
  aux::PrintArgv(argc, argv);

  // Save the number of input arguments
  const int NARGC = argc - 1;

  // Create generator object first
  std::unique_ptr<MGraniitti> gen = std::make_unique<MGraniitti>();

  try {
    cxxopts::Options options(argv[0], "");

    options.add_options("")("i,INPUT", "Input card", cxxopts::value<std::string>())("H,help",
                                                                                    "Help");

    options.add_options("GENERALPARAM")

        ("o,OUTPUT", "Output name             <string>", cxxopts::value<std::string>())(
            "f,FORMAT", "Output format          <hepmc3|hepmc2|hepevt>",
            cxxopts::value<std::string>())("n,NEVENT", "Number of events       <integer32>",
                                           cxxopts::value<unsigned int>())(
            "g,INTEGRATOR", "Integrator             <VEGAS|FLAT>", cxxopts::value<std::string>())(
            "w,WEIGHTED", "Weighted events        <true|false>", cxxopts::value<std::string>())(
            "c,CORES", "Number of CPU threads  <integer>", cxxopts::value<unsigned int>());

    options.add_options("PROCESSPARAM")("p,PROCESS", "Process                <string>",
                                        cxxopts::value<std::string>())(
        "e,ENERGY", "CMS energy              <double>", cxxopts::value<double>())(
        "l,POMLOOP", "Screening Pomeron loop  <true|false>", cxxopts::value<std::string>())(
        "s,NSTABS", "Excite protons          <0|1|2>", cxxopts::value<unsigned int>())(
        "q,LHAPDF", "Set LHAPDF              <string>", cxxopts::value<std::string>())(
        "h,HIST", "Histogramming           <0|1|2>", cxxopts::value<unsigned int>())(
        "r,RNDSEED", "Random seed            <integer32>", cxxopts::value<unsigned int>());

    auto r = options.parse(argc, argv);

    if (r.count("help") || NARGC == 0) {
      gen->GetProcessNumbers();
      std::cout << options.help({"", "GENERALPARAM", "PROCESSPARAM"}) << std::endl;
      std::cout << rang::style::bold << " Arrow operators:" << rang::style::reset << std::endl;
      std::cout << "  use -> between initial and final states" << std::endl;
      std::cout << "  use &> (instead of ->) for a decoupled central system phase space (use with "
                   "<P>, <P> class, e.g. for s-channel resonances)"
                << std::endl;
      std::cout
          << "  use  > for recursive decaytree branchings with curly brackets { grand daughters }"
          << std::endl;
      std::cout << std::endl;
      std::cout << rang::style::bold
                << " Inline 'on-the-flight' parameters to concatenate with PROCESS string:"
                << rang::style::reset << std::endl
                << std::endl;

      std::cout << rang::style::bold << "  [Generic]" << rang::style::reset << std::endl;

      std::cout << "  @FLATAMP:N                                  flat matrix element for 'pure' phase "
```

./src/Program/gr.cc 2/4

```
                   "space generation, set N to -1 for more info"
                << std::endl;
      std::cout << "  @FLATMASS2:true                             flat sampling in M^2 instead of "
                   "relativistic Breit-Wigner f(M^2) in decay trees"
                << std::endl;
      std::cout << "  @OFFSHELL:X                                 how many +- full widths particles "
                   "off-shell in decay trees (X = 0 on-shell, X = 5 default)"
                << std::endl;
      std::cout
          << "  @PDG[X](M:350.0, W:5.0)                     new mass and width for pdg particle id X"
          << std::endl;
      std::cout
          << "  @R[f0_980](M:0.990, W:0.065)                set new central resonance mass and width"
          << std::endl;
      std::cout << "  @RES{rho_770:1, ..., f2_1270:1}         set active resonances in the "
                   "amplitude (1 active, 0 inactive)"
                << std::endl;

      std::cout << std::endl;
      std::cout << rang::style::bold << "  [Pomeron amplitudes]" << rang::style::reset << std::endl;

      std::cout << "  @SPINGEN:true                               set generation 2->1 spin "
                   "correlations active (true, false)"
                << std::endl;
      std::cout << "  @SPINDEC:true                               set decay 1->2 spin correlations "
                   "active (true, false)"
                << std::endl;

      std::cout << "  @FRAME:X                                    set Lorentz frame definition for the "
                   "spin density matrix (X = CS, HX, CM)"
                << std::endl;
      std::cout << "  @R[f2_1270]{JZ0:0.5, JZ1:0.0, JZ2:0.5}  set new diagonal spin density matrix "
                   "elements for resonances"
                << std::endl;
      std::cout << "  @JMAX:X                                     set maximum Pomeron helicity for the "
                   "sliding helicity amplitudes"
                << std::endl;

      std::cout << std::endl;
      std::cout << rang::style::bold << "  [Tensor pomeron amplitudes]" << rang::style::reset
                << std::endl;

      std::cout << "  @R[f0_980]{g0:1.0, g1:0.2, ...}             set new production couplings {g0,g1} "
                   "[scalar/pseudoscalar] {g0,...,g6} [tensor]"
                << std::endl;

      std::cout << std::endl;
      std::cout << rang::style::bold << " PROCESS string examples:" << rang::style::reset
                << std::endl;

      std::cout << "  yy[CON]<C> -> mu+ mu-" << std::endl;
      std::cout << "  yy[Higgs]<F> &> 22 22" << std::endl;
      std::cout << "  yy[CON]<C> -> 992 -992 @PDG[992](M:1500,W:0)" << std::endl;
      std::cout << "  yP[RESTENSOR]<C> -> pi+ pi- @RES{rho_770:1}" << std::endl;
      std::cout << "  PP[CON]<C> -> rho(770)0 > (pi+ pi-) rho(770)0 > (pi+ pi-)" << std::endl;
      std::cout << "  PP[CON]<F> -> pi+ pi- pi+ pi-" << std::endl;
      std::cout << "  PP[CON]<C> -> p+ p- @FLATAMP:2" << std::endl;
      std::cout << "  PP[RES+CON]<F> -> pi+ pi- @RES{f0_500:0,rho_770:1,f0_980:1,f2_1270:1} "
                   "@R[f0_980](M:0.98,W:0.065)"
                << std::endl;

      std::cout << std::endl;

      std::cout << std::endl;
      std::cout << rang::style::bold << rang::fg::green
                << " A steering card example with no additional input:" << rang::style::reset
                << std::endl;
      std::cout << "  " << argv[0] << " -i ./input/test.json" << std::endl << std::endl;

      std::cout << rang::style::bold << rang::fg::red
                << " A steering card example with commandline input override:" << rang::style::reset
                << std::endl;
      std::cout << "  " << argv[0] << " -i ./input/test.json -p \"yy[QED]<F> -> a+ a-\""
                << std::endl
                << std::endl;

      aux::CheckUpdate();

      return EXIT_FAILURE;
    }

    // ------------------------------------------------------------------
    // This first
    // Process re-set from commandline
```

./src/Program/gr.cc 3/4

```
    if (r.count("p")) {
      gen->ReadInput(r["i"].as<std::string>(), r["p"].as<std::string>());
    } else {
      gen->ReadInput(r["i"].as<std::string>());
    }

    // ------------------------------------------------------------------
    // Quick custom parameters

    // General parameters (order is important)
    if (r.count("n")) { gen->SetNumberOfEvents(r["n"].as<unsigned int>()); }
    if (r.count("o") && r.count("f")) {
      gen->SetOutput(r["o"].as<std::string>());
      gen->SetFormat(r["f"].as<std::string>());
    } else if ((r.count("o") && !r.count("f")) || (!r.count("o") && r.count("f"))) {
      throw std::invalid_argument("Error: Use options -o and -f together.");
    }
    if (r.count("g")) {
      const std::string val = r["g"].as<std::string>();
      gen->SetIntegrator(val);
    }
    if (r.count("w")) {
      const std::string val = r["w"].as<std::string>();
      gen->SetWeighted(val == "true");
    }
    if (r.count("c")) { gen->SetCores(r["c"].as<unsigned int>()); }

    // Process parameters (adding more might be involved due to initialization
    // in
    // MProcess...)
    if (r.count("l")) {
      const std::string val = r["l"].as<std::string>();
      gen->proc->SetScreening(val == "true");
    }
    if (r.count("s")) {
      double E = r["e"].as<double>() / 2.0;  // CMS-energy in symmetric collider
      gen->proc->SetBeamEnergies(E, E);
    }
    if (r.count("s")) { gen->proc->SetExcitation(r["s"].as<unsigned int>()); }
    if (r.count("q")) { gen->proc->SetLHAPDF(r["q"].as<std::string>()); }
    if (r.count("h")) { gen->proc->SetHistograms(r["h"].as<unsigned int>()); }
    if (r.count("r")) { gen->proc->random.SetSeed(r["r"].as<unsigned int>()); }

    // ------------------------------------------------------------------

    // Initialize generator (always last!)
    gen->Initialize();

    // Generate events
    gen->Generate();

    // Print histograms
    gen->PrintHistograms();

    // Exception handling
  } catch (const std::invalid_argument &e) {
    gen->GetProcessNumbers();
    gra::aux::PrintGameOver();
    std::cerr << rang::fg::red << "Exception catched: " << rang::fg::reset << e.what() << std::endl;
    return EXIT_FAILURE;
  } catch (const std::ios_base::failure &e) {
    gra::aux::PrintGameOver();
    std::cerr << rang::fg::red << "Exception catched: std::ios_base::failure: " << rang::fg::reset
              << e.what() << std::endl;
    return EXIT_FAILURE;
  } catch (const cxxopts::OptionException &e) {
    gra::aux::PrintGameOver();
    std::cerr << rang::fg::red << "Exception catched: Commandline options: " << rang::fg::reset
              << e.what() << std::endl;
    return EXIT_FAILURE;
  } catch (const nlohmann::json::exception &e) {
    gra::aux::PrintGameOver();
    std::cerr << rang::fg::red << "Exception catched: JSON input: " << rang::fg::reset << e.what()
              << std::endl;
    return EXIT_FAILURE;
  } catch (...) {
    gen->GetProcessNumbers();
    gra::aux::PrintGameOver();
    std::cerr << rang::fg::red << "Exception catched: Unspecified (...) (Probably JSON input)"
              << rang::fg::reset << std::endl;
    return EXIT_FAILURE;
  }
```

./src/Program/gr.cc 4/4

```
  std::cout << "[gr: done]" << std::endl;
  aux::CheckUpdate();

  return EXIT_SUCCESS;
}
```

./src/MUserHistograms.cc **1/2**

```
 1: // Container class for different type of histograms
 2: //
 3: // (c) 2017-2020 Mikael Mieskolainen
 4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
 5:
 6: // C++
 7: #include <complex>
 8: #include <iostream>
 9: #include <vector>
10:
11: // Own
12: #include "Graniitti/MAux.h"
13: #include "Graniitti/MForm.h"
14: #include "Graniitti/MKinematics.h"
15: #include "Graniitti/MMath.h"
16: #include "Graniitti/MSpin.h"
17: #include "Graniitti/MUserHistograms.h"
18:
19: using gra::aux::indices;
20: using gra::math::PI;
21:
22: namespace gra {
23: void MUserHistograms::InitHistograms() {
24:   // Level 1
25:   unsigned int Nbins = 40;
26:
27:   h1["M"]    = MH1<double>(Nbins, "Central System M (GeV)");
28:   h1["Rap"] = MH1<double>(Nbins, "Central System Rap");
29:   h1["Rap"].SetAutoSymmetry(true);
30:   h1["Pt"]       = MH1<double>(Nbins, 0.0, 2.5, "Central System Pt (GeV)");
31:   h1["dPhi_pp"]  = MH1<double>(Nbins, 0.0, gra::math::PI, "Forward deltaphi (rad)");
32:   h1["pPt"]      = MH1<double>(Nbins, 0.0, 2.0, "Forward Pt (GeV)");
33:   h2["raplrap2"] = MH2(Nbins, Nbins, "Rapidity1 vs Rapidity2");
34:   h2["raplrap2"].SetAutoSymmetry((true, true));
35:
36:   h1["FM"] = MH1<double>(Nbins, "Forward M (GeV)");
37:   h1["m0"] = MH1<double>(Nbins, "Intermediate daughter M (GeV)");
38:
39:   // Level 2
40:   Nbins = 40;
41:
42:   h2["costhetaphi_CS"] =
43:       MH2(Nbins, -1.0, 1.0, Nbins, -PI, PI, "(cos theta, phi) [CS] [Collins-Soper frame]");
44:   h2["costhetaphi_HX"] =
45:       MH2(Nbins, -1.0, 1.0, Nbins, -PI, PI, "(cos theta, phi) [HX] [Helicity frame]");
46:   h2["costhetaphi_AH"] =
47:       MH2(Nbins, -1.0, 1.0, Nbins, -PI, PI, "(cos theta, phi) [AH] [Anti-Helicity frame]");
48:   h2["costhetaphi_PG"] =
49:       MH2(Nbins, -1.0, 1.0, Nbins, -PI, PI, "(cos theta, phi) [PG] [Pseudo-GJ frame]");
50:   h2["costhetaphi_GJ"] =
51:       MH2(Nbins, -1.0, 1.0, Nbins, -PI, PI, "(cos theta, phi) [GJ] [Gottfried-Jackson frame]");
52:   h2["costhetaphi_CM"] =
53:       MH2(Nbins, -1.0, 1.0, Nbins, -PI, PI, "(cos theta, phi) [CM] [Direct system rest frame]");
54:
55:   h2["costhetaphi_LA"] =
56:       MH2(Nbins, -1.0, 1.0, Nbins, -PI, PI, "(cos theta, phi) [LAB] [Laboratory frame]");
57: }
58:
59: // Input as the total event weight
60: void MUserHistograms::FillHistograms(double totalweight, const gra::LORENTZSCALAR &lts) {
61:   // Level 1
62:   if (HIST >= 1) {
63:     h1["M"].Fill(gra::math::msqrt(lts.m2), totalweight);
64:     h1["Rap"].Fill(lts.Y, totalweight);
65:     h1["Pt"].Fill(lts.Pt, totalweight);
66:     h1["dPhi_pp"].Fill(std::abs(lts.pfinal[1].DeltaPhi(lts.pfinal[2])), totalweight);
67:     h1["pPt"].Fill(lts.pfinal[1].Pt(), totalweight);
68:
69:     // Cascade decay
70:     if (lts.decaytree[0].legs.size() > 0) { h1["m0"].Fill(lts.decaytree[0].p4.M(), totalweight); }
71:
72:     // Dissociated proton
73:     if (lts.pfinal[1].M() > 1.0) { h1["FM"].Fill(lts.pfinal[1].M(), totalweight); }
74:   }
75:
76:   // Level 2
77:   if (HIST >= 2) {
78:     h2["raplrap2"].Fill(lts.decaytree[0].p4.Rap(), lts.decaytree[1].p4.Rap(), totalweight);
79:     FillCosThetaPhi(totalweight, lts);
80:   }
81: }
82:
83: // (costheta,phi) of daughter in different Lorentz frames, input as the total
```

./src/MUserHistograms.cc **2/2**

```
84: // event weight
85: void MUserHistograms::FillCosThetaPhi(double totalweight, const gra::LORENTZSCALAR &lts) {
86:   // Central final states
87:   std::vector<M4Vec> pf;
88:   pf.push_back(lts.decaytree[0].p4);
89:   pf.push_back(lts.decaytree[1].p4);
90:
91:   // System 4-momentum
92:   M4Vec X;
93:   for (const auto &i : indices(lts.decaytree)) { X += lts.decaytree[i].p4; }
94:
95:   // ------------------------------------------------------------------
96:   // Choose Pseudo-Gottfried-Jackson beam direction (-1,1)
97:   const int direction = 1;
98:
99:   {
100:     // ** PREPARE LORENTZ TRANSFORMATION COMMON VARIABLES **
101:     M4Vec           pb1boost;   // beam1 particle boosted
102:     M4Vec           pb2boost;   // beam2 particle boosted
103:     std::vector<M4Vec> pfboost;   // central particles boosted
104:     gra::kinematics::LorentFramePrepare(pf, X, lts.pbeam1, lts.pbeam2, pb1boost, pb2boost, pfboost);
105:
106:     // ** TRANSFORM TO DIFFERENT LORENTZ FRAMES **
107:     std::vector<std::string> frametype = {"CS", "HX", "AH", "PG", "CM"};
108:     for (const auto &k : indices(frametype)) {
109:       // Transform and histogram
110:       std::vector<M4Vec> pfout;
111:       gra::kinematics::LorentzFrame(pfout, pb1boost, pb2boost, pfboost, frametype[k], direction);
112:       h2["costhetaphi_" + frametype[k]].Fill(std::cos(pfout[0].Theta()), pfout[0].Phi(),
113:                                              totalweight);
114:       // h2["Mphi_"+frametype[k]].Fill(gra::math::msqrt(lts.m2), pfout[0].Phi(),
115:       // totalweight);
116:     }
117:   }
118:
119:   // ------------------------------------------------------------------
120:   // Gottfried-Jackson frame
121:   std::vector<M4Vec> pfGJ = pf;
122:   gra::kinematics::GJframe(pfGJ, lts.pfinal[0], direction, lts.q1, lts.q2, false);
123:   h2["costhetaphi_GJ"].Fill(std::cos(pfGJ[0].Theta()), pfGJ[0].Phi(), totalweight);
124:
125:   // Laboratory frame
126:   h2["costhetaphi_LA"].Fill(std::cos(pf[0].Theta()), pf[0].Phi(), totalweight);
127: }
128:
129: // Print all histograms out
130: void MUserHistograms::PrintHistograms() {
131:   if (HIST >= 1) {
132:     for (auto const &x : h1) { h1[x.first].Print(); }
133:   }
134:   if (HIST >= 2) {
135:     for (auto const &x : h2) { h2[x.first].Print(); }
136:   }
137: }
138:
139: }  // namespace gra
140:
```

./src/MSudakov.cc **1/8**

```
 1: // Shuvaev PDF and Sudakov suppression class
 2: //
 3: // (c) 2017-2020 Mikael Mieskolainen
 4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
 5:
 6: // C++
 7: #include <complex>
 8: #include <fstream>
 9: #include <future>
10: #include <iostream>
11: #include <memory>
12: #include <vector>
13:
14: // C file processing
15: #include <sys/stat.h>
16: #include <sys/types.h>
17: #include <unistd.h>
18:
19: // Own
20: #include "Graniitti/MAux.h"
21: #include "Graniitti/MMath.h"
22: #include "Graniitti/MSudakov.h"
23: #include "Graniitti/MTimer.h"
24:
25: // LHAPDF
26: #include "LHAPDF/LHAPDF.h"
27:
28: // Libraries
29: #include "rang.hpp"
30:
31: namespace gra {
32:
33: using aux::indices;
34: using math::msqrt;
35: using math::pow2;
36: using math::zi;
37:
38: // Constructor
39: MSudakov::MSudakov() {}
40:
41: // Destructor
42: MSudakov::~MSudakov() { delete PdfPtr; }
43:
44: // Init
45: void MSudakov::Init(double _sqrts, const std::string &PDFSET, bool init_arrays) {
46:   // First this
47:   Numerics.ReadParameters();
48:
49:   // Setup energy dependent boundaries
50:   Numerics.sqrts = _sqrts;
51:   Numerics.M_MAX = _sqrts;
52:
53:   // Collinear case: M^2 = x_1x_2s, let x_1 = 1.0 gives minimum x_2 = M^2/s
54:   Numerics.x_MIN = math::pow2(Numerics.M_MIN / Numerics.sqrts);
55:
56:   // InitLHAPDF("CT10nlo");
57:   InitLHAPDF(PDFSET);
58:
59:   if (init_arrays) { InitArrays(); }
60: }
61:
62: void MSudakov::InitArrays() {
63:   // Init Sudakov variables
64:   {
65:     std::cout << "Initializing <Sudakov> array:" << std::endl;
66:
67:     // q2,M
68:     veto.sqrts = Numerics.sqrts;  // FIRST THIS
69:     veto.Set(0, "q2", Numerics.q2_MIN, Numerics.q2_MAX, Numerics.SUDA_N[0],
70:              Numerics.SUDA_log_ON[0]);
71:     veto.Set(1, "M", Numerics.M_MIN, Numerics.M_MAX, Numerics.SUDA_N[1], Numerics.SUDA_log_ON[1]);
72:     veto.InitArray();  // Initialize (call last!)
73:
74:     const unsigned long hash     = gra::aux::djb2hash(veto.GetHashString());
75:     const std::string   filename = gra::aux::GetBasePath(2) + "/eikonal/SUDA_" +
76:                                    gra::aux::ToString(veto.sqrts, 0) + "_" + PDFSETNAME + "_" +
77:                                    std::to_string(hash);
78:
79:     // Try to read pre-calculated
80:     bool ok = veto.ReadArray(filename);
81:
82:     while (!ok) {  // Problem, re-calculate
83:       // Pointer to member function: ReturnType
```

./src/MSudakov.cc **2/8**

```
84:       // (ClassType::*)(ParameterTypes...)
85:       std::pair<double, double> (MSudakov::*f)(double, double) = &MSudakov::Sudakov_T;
86:       CalculateArray(veto, f);
87:       veto.WriteArray(filename, true);
88:       ok = veto.ReadArray(filename);
89:     }
90:   }
91:
92:   // Init Shuvaev PDF variables
93:   {
94:     std::cout << "Initializing <Shuvaev> array:" << std::endl;
95:
96:     // q2,x
97:     spdf.sqrts = Numerics.sqrts;  // FIRST THIS
98:     spdf.Set(0, "q2", Numerics.q2_MIN, Numerics.q2_MAX, Numerics.SHUV_N[0],
99:              Numerics.SHUV_log_ON[0]);
100:     spdf.Set(1, "x", Numerics.x_MIN, Numerics.x_MAX, Numerics.SHUV_N[1], Numerics.SHUV_log_ON[1]);
101:     spdf.InitArray();  // Initialize (call last!)
102:
103:     const unsigned long hash     = gra::aux::djb2hash(spdf.GetHashString());
104:     const std::string   filename = gra::aux::GetBasePath(2) + "/eikonal/SHUV_" +
105:                                    gra::aux::ToString(spdf.sqrts, 0) + "_" + PDFSETNAME + "_" +
106:                                    std::to_string(hash);
107:
108:     // Try to read pre-calculated
109:     bool ok = spdf.ReadArray(filename);
110:
111:     while (!ok) {  // Problem, re-calculate
112:       // Pointer to member function: ReturnType
113:       // (ClassType::*)(ParameterTypes...)
114:       std::pair<double, double> (MSudakov::*f)(double, double) = &MSudakov::Shuvaev_R;
115:       CalculateArray(spdf, f);
116:       spdf.WriteArray(filename, true);
117:       ok = spdf.ReadArray(filename);
118:     }
119:   }
120:   // ------------------------------------------------------
121:   // Finally
122:   initialized = true;
123:   std::cout << std::endl;
124:
125:   if (Numerics.DEBUG) { TestPDF(); }
126: }
127:
128: // Return gluon xg(x,Q2) from LHAPDF
129: double MSudakov::xg_xQ2(double x, double q2) const {
130:   const int pid = 21;  // gluon
131:   try {
132:     return PdfPtr->xfxQ2(pid, x, q2);
133:   } catch (...) {
134:     std::string str =
135:         "MSudakov::xg_xQ2: Problem with x = " + std::to_string(x) + ", q2 = " + std::to_string(q2);
136:     throw std::invalid_argument(str);
137:   }
138: }
139:
140: // PDF access
141: //
142: // [REFERENCE: LHAPDF6, arxiv.org/abs/1412.7420]
143: void MSudakov::InitLHAPDF(const std::string &pdfname) {
144:   PDFSETNAME = pdfname;
145:
146:   // LHAPDF init
147:   try {
148:     PdfPtr = LHAPDF::mkPDF(pdfname, 0);
149:   } catch (...) {
150:     ++init_trials;
151:
152:     std::string str = "MSudakov::InitLHAPDF: Trials = " + std::to_string(init_trials) +
153:                       " :: Problem with reading a pdfset '" + pdfname + "'";
154:
155:     if (init_trials >= 2) {  // Too many failures
156:       throw std::invalid_argument(str);
157:     }
158:
159:     std::cout << str << std::endl;
160:
161:     aux::AutoDownloadLHAPDF(pdfname);
162:     InitLHAPDF(pdfname);  // try again
163:   }
164: }
165:
166: // PDF test routine
```

```
167: void MSudakov::TestPDF() const {
168:   const double MINLOGX = std::log10(Numerics.x_MIN);
169:   const double MAXLOGX = std::log10(Numerics.x_MAX);
170:   const int    NX      = 5;  // Number of points - 1
171:   const double stepX   = (MAXLOGX - MINLOGX) / NX;
172:
173:   const double MINLOGQ2 = std::log10(Numerics.q2_MIN);
174:   const double MAXLOGQ2 = std::log10(Numerics.q2_MAX);
175:   const int    NQ2      = 5;  // Number of points - 1
176:   const double stepQ2   = (MAXLOGQ2 - MINLOGQ2) / NQ2;
177:
178:   const double MINLOGM = std::log10(Numerics.M_MIN);
179:   const double MAXLOGM = std::log10(Numerics.M_MAX);
180:   const int    NM      = 5;  // Number of points - 1
181:   const double stepM   = (MAXLOGM - MINLOGM) / NM;
182:
183:   // Test loop
184:   for (std::size_t i = 0; i < NM + 1; ++i) {
185:     const double log10M = MINLOGM + i * stepM;
186:     const double M      = std::pow(10, log10M);
187:
188:     printf("[M = %0.1f GeV] : alpha_s(Q = M GeV) = %0.3f \n\n", M, PdfPtr->alphasQ2(M * M));
189:
190:     for (std::size_t j = 0; j < NX + 1; ++j) {
191:       const double log10x = MINLOGX + j * stepX;
192:       const double x      = std::pow(10, log10x);
193:
194:       printf("x = %0.5E \n", x);
195:       for (std::size_t k = 0; k < NQ2 + 1; ++k) {
196:         const double log10q2 = MINLOGQ2 + k * stepQ2;
197:         const double q2      = std::pow(10, log10q2);
198:
199:         // Normal gluon pdf
200:         const double xf = xg_xQ2(x, q2);
201:
202:         // Durham flux
203:         const double hxf = fg_xQ2M(x, q2, M);
204:
205:         printf(
206:             "[x = %0.3E, q2 = %0.2f, M = %0.1f) : [gluon pdf: xg(x,q2), "
207:             "Durham flux: fg(x,q2,M)] = (%0.2f,%0.2f) \n",
208:             x, q2, M, xf, hxf);
209:       }
210:       std::cout << std::endl;
211:     }
212:     std::cout << std::endl;
213:   }
214: }
215:
216: // Access QCD coupling alpha_s(Q^2) from LHAPDF
217: double MSudakov::AlphaS_Q2(double q2) const {
218:   try {
219:     return PdfPtr->alphasQ2(q2);
220:   } catch (...) {
221:     std::string str = "MSudakov::AlphaS_Q2: Problem with q2 = " + std::to_string(q2);
222:     throw std::invalid_argument(str);
223:   }
224: }
225:
226: // Calculate differentiation dxg/dQ2 via "Richardson's extrapolation"
227: // f'(x) = [4*D_0(h) + D_0(2h)] / 3 + O(h^4)
228: //
229: // Requires 4 evaluations of densities
230: //
231: // [REFERENCE: en.wikipedia.org/wiki/Richardson_extrapolation]
232: double MSudakov::diff_xg_xQ2_wrt_Q2(double x, double q2) const {
233:   // Do not take h less than 1E-4 with 64-bit double
234:   const double h   = 1E-4;
235:   const double hX2 = 2 * h;
236:
237:   // Calculate central differences
238:   const double D0_A = (xg_xQ2(x, q2 + h) - xg_xQ2(x, q2 - h)) / (2 * h);
239:   const double D0_B = (xg_xQ2(x, q2 + hX2) - xg_xQ2(x, q2 - hX2)) / (2 * hX2);
240:
241:   return (4.0 * D0_A - D0_B) / 3.0;
242: }
243:
244: // Durham flux (skewed gluon pdf)
245: // ~ Shuvaev transformed gluon pdf x Sudakov suppression
246: //
247: // f_g(x,x',qt^2,\mu)
248: // = \frac{\partial}{\partial \ln Q_t^2} [R_g(x/2,x/2,Q_t^2)\sqrt{T(Q_t^2,\mu)}]
249: //
```



```
250: double MSudakov::fg_xQ2M(double x, double q2, double M) const {
251:   // Calculate Shuvaev transformation
252:   // std::pair<double,double> out1 = Shuvaev_H(q2, x);
253:   std::pair<double, double> out1 = spdf.Interpolate2D(q2, x);
254:   const double              Hg   = out1.first;
255:   const double              dHg  = out1.second;
256:
257:   // Calculate Sudakov veto
258:   // std::pair<double,double> out2 = Sudakov_T(q2, M);
259:   std::pair<double, double> out2 = veto.Interpolate2D(q2, M);
260:   const double              Tg   = out2.first;
261:   const double              dTg  = out2.second;
262:
263:   // Chain rule's: d/dln(q^2) [ ... ]
264:   double total = 0.0;
265:
266:   if (Tg > 1e-15) {
267:     total = dHg * math::msqrt(Tg) + Hg * dTg / (2.0 * math::msqrt(Tg));
268:   } else {
269:     // printf("MSudakov::fg_xQ2M: Sudakov factor Tg = %0.2E [x = %0.1E, q2 =
270:     //  %0.3f
271:     //  GeV^2, M = %0.3f GeV] \n", Tg, x, q2, M);
272:   }
273:
274:   // And because our functions return partial derivatives w.r.t. q^2 (not ln q^2)
275:   // we transform:
276:   // d(ln(q^2)/dq^2 = 1/q^2 <=> d(ln(q^2) = dq^2/q^2 =>
277:   // d[...]/d(ln(q^2) = d[...]/dq^2 * q^2, applied below:
278:   //
279:   total *= q2;
280:
281:   // Truncate negative values to give proper zero
282:   total = (total < 0.0) ? 0.0 : total;
283:
284:   return total;
285: }
286:
287: // Calculate Shuvaev integral transform
288: // ------------------------------------------------------------------
289: // Identity:
290: //
291: // H_g(x, \xi -> 0) = xg(x)
292: //
293: // ------------------------------------------------------------------
294: //
295: // Hg  = numerical integral transformed from standard pdf
296: // dHg = dHg/dq^2 (differentiated numerically)
297: //
298: // [REFERENCE: Harland-Lang, arxiv.org/abs/1306.6661]
299: //
300: std::pair<double, double> MSudakov::ShuvaevH(double q2, double x) {
301:   const double y_MIN  = x / 4.0;
302:   const double y_MAX  = 1.0;
303:   const double y_STEP = (y_MAX - y_MIN) / Numerics.ShuvaevIntegralN;
304:
305:   double Hg  = 0.0;
306:   double dHg = 0.0;
307:
308:   // Check that we are within valid domain (take into account floating points)
309:   const double EPS = 1e-5;
310:   if (x >= Numerics.x_MIN * (1 - EPS) && x <= Numerics.x_MAX * (1 + EPS) &&
311:       q2 >= Numerics.q2_MIN * (1 - EPS) && q2 <= Numerics.q2_MAX * (1 + EPS)) {
312:     // N+1!
313:     std::vector<double> fA(Numerics.ShuvaevIntegralN + 1, 0.0);
314:     std::vector<double> fB(Numerics.ShuvaevIntegralN + 1, 0.0);
315:
316:     for (const auto &i : indices(fA)) {
317:       const double y        = y_MIN + i * y_STEP;
318:       const double argument = x / (4.0 * y);
319:
320:       // H_g(x/2,x/2,Q^2) = 4x/\pi \int_{x/4}^1 dy
321:       // y^{1/2}(1-y)^{1/2}g(x/4y),q^2)
322:       //
323:       // => take into account that LHAPDF provides xg(), not g(), gives:
324:       const double factor = math::msqrt(math::pow3(y) * (1 - y));
325:       fA[i]               = factor * xg_xQ2(argument, q2);
326:       fB[i]               = factor * diff_xg_xQ2_wrt_Q2(argument, q2);
327:     }
328:     const double norm = 16.0 / math::PI;
329:     Hg                = norm * math::CSIntegral(fA, y_STEP);
330:     dHg               = norm * math::CSIntegral(fB, y_STEP);
331:   } else {
332:     // Fatal error
```



```
333:     throw std::invalid_argument("MSudakov::Shuvaev_H(q2,x): Input arguments out of domain: q2 = " +
334:                                 std::to_string(q2) + ", x = " + std::to_string(x));
335:   }
336:   return (Hg, dHg);
337: }
338:
339: // Calculate Sudakov real radiation suppression (veto) factor via direct
340: // numerical integral.
341: //
342: // T(Q_t^2, \mu) = exp( -\int_{Q_t^2}^{\mu^2} dk_t^2 \frac{1}{k_t^2}
343: // \frac{\alpha_s(k_t^2)}{2\pi}
344: //                      \int_0^{1-DELTA} dz[zP_gg(z) + \sum_q P_qg(z)] ),
345: //
346: // where DELTA = k_t/mu
347: //
348: // Tg  = numerical integral value [range [0,1)]
349: // dTg = dTg/dq^2 (analytic derivative)
350: //
351: // [REFERENCE: Coughlin, Forshaw, arxiv.org/abs/0912.3280v2]
352: //
353: std::pair<double, double> MSudakov::Sudakov_T(double qt2, double mu) {
354:   // Scale functor
355:   auto DeltaScale = [](double kt2, double M) -> double { return math::msqrt(kt2) / M; };
356:
357:   // [OUTER INTEGRAL]: k_t^2 integral upper and lower bounds:
358:   const double MAX = std::log(math::pow2(mu));
359:   const double MIN = std::log(qt2);
360:
361:   // Discretization
362:   const double STEP = (MAX - MIN) / Numerics.SudakovIntegralN;
363:
364:   // [OUTER kt^2-integral]:
365:   // Change of a variable: u := ln(kt^2) [du = (du/dkt2) dkt2]
366:   // this cancels 1/kt^2 from the integrand and numerical convergence is
367:   // much improved.
368:   //
369:   // [INNER z-integral is analytic]:
370:   std::vector<double> f(Numerics.SudakovIntegralN + 1, 0.0);
371:   for (const auto &i : indices(f)) {
372:     const double u     = MIN + i * STEP;
373:     const double kt2   = std::exp(u);
374:     const double delta = DeltaScale(kt2, mu);
375:     f[i]               = AlphaS_Q2(kt2) * (AP_gg(delta) + AP_qg(delta, kt2));
376:   }
377:   const double integral = math::CSIntegral(f, STEP) / (2.0 * math::PI);
378:
379:   // Final expression and its derivative
380:   const double Tg   = std::exp(-integral);
381:   const double delta = DeltaScale(qt2, mu);
382:   const double dTg =
383:       Tg * AlphaS_Q2(qt2) / (2.0 * math::PI * qt2) * (AP_gg(delta) + AP_qg(delta, qt2));
384:   return (Tg, dTg);
385: }
386:
387: // Altarelli-Parisi splitting function definite integral over z
388: //
389: // see standard literature, e.g.
390: // [REFERENCE: B.R. Webber, CERN lectures 08,
391: // www.hep.phy.cam.ac.uk/theory/webber/QCDlect3.pdf]
392: // [REFERENCE: R.K. Ellis, W.J. Stirling, B.R. Webber, QCD and Collider Physics]
393: // [REFERENCE: http://pdg.lbl.gov/2018/download/db2018.pdf]
394: //
395: // \int z P_gg(z) dz
396: // \int z *( 2 * CA * int [z/(1-z)_(+) + (1-z)/z + z*(1-z)] +
397: //           1/6*(11*CA - 4*Nf*TR)*deltafunc(1-z) ) dz, z = 0 ... (1-delta) =
398: //                    (below)
399: //
400: // First part: ...
401: // Second part: \int 1/6*(11*C - 4*N*T)*DiracDelta[1-x] dx, x = 0 ... 1 - delta,
402: // this vanishes with positive delta
403: //
404: // where + is the "plus-description"
405: //
406: double MSudakov::AP_gg(double delta) const {
407:   const double CA = 3.0;  // Structure constant
408:   return 2.0 * CA *
409:          (std::log(1.0 / delta) -
410:           math::pow2(1.0 - delta) * (3.0 * math::pow2(delta) - 2.0 * delta + 11.0) / 12.0);
411: }
412:
413: // Altarelli-Parisi splitting function definite integral over z,
414: //
415: // \int \sum_q P_qz(z) dz
```



```
416: //
417: // with sum over active quark flavors with mass threshold gt2
418: // \sum_q TR * \int (z^2 + (1-z)^2) dz, z = 0 ... (1-delta)
419: //
420: double MSudakov::AP_qg(double delta, double qt2) const {
421:   const double TR = 0.5;  // Structure constant
422:   return TR * (-2.0 * math::pow3(delta) / 3.0 + math::pow2(delta) - delta + 2.0 / 3.0) *
423:          NumFlavor(qt2);
424: }
425:
426: // Return the number of quark flavors at scale q^2
427: //
428: double MSudakov::NumFlavor(double q2) const {
429:   const double m_charm  = 1.275;  // Gev, PDG-2018 (default definition)
430:   const double m_bottom = 4.18;   // Gev
431:
432:   if (q2 < math::pow2(m_charm)) {
433:     return 3.0;
434:   } else if (q2 < math::pow2(m_bottom)) {
435:     return 4.0;
436:   } else {
437:     return 5.0;
438:   }
439: }
440:
441: // Constructs interpolation array values
442: void MSudakov::CalculateArray(IArray2D &arr,
443:                               std::pair<double, double> (MSudakov::*f)(double, double)) {
444:   MTimer timer;
445:   for (const auto &i : indices(arr.F)) {
446:     const double a = arr.MIN[0] + i * arr.STEP[0];
447:
448:     // Transform input to linear if log stepping, for the function
449:     const double var1 = (arr.islog[0]) ? std::exp(a) : a;
450:     gra::aux::PrintProgress(i / static_cast<double>(arr.N[0] + 1));
451:
452:     for (const auto &j : indices(arr.F[i])) {
453:       const double b = arr.MIN[1] + j * arr.STEP[1];
454:
455:       // Transform input to linear if log stepping, for the function
456:       const double var2 = (arr.islog[1]) ? std::exp(b) : b;
457:
458:       // Call function being pointed to
459:       const std::pair<double, double> output = (this->*f)(var1, var2);
460:
461:       arr.F[i][j][0] = a;
462:       arr.F[i][j][1] = b;
463:       arr.F[i][j][2] = std::abs(output.first) < 1e-64 ? 0 : output.first;    // Underflow protection
464:       arr.F[i][j][3] = std::abs(output.second) < 1e-64 ? 0 : output.second;  //
465:     }
466:   }
467:   // Progressbar clearing
468:   gra::aux::ClearProgress();
469:   printf("MSudakov::CalculateArray: Time elapsed %0.1f sec \n", timer.ElapsedSec());
470: }
471:
472: // Write the array to a file
473: bool IArray2D::WriteArray(const std::string &filename, bool overwrite) const {
474:   // Do not write if file exists already
475:   if (gra::aux::FileExist(filename) && !overwrite) {
476:     // std::cout << "- Found pre-calculated" << std::endl;
477:     return true;
478:   }
479:   // ------------------------------------------------------------------
480:
481:   std::ofstream file;
482:   file.open(filename);
483:   if (!file.is_open()) {
484:     std::string str = "IArray2D::WriteArray: Fatal IO-error with: " + filename;
485:     throw std::invalid_argument(str);
486:   }
487:
488:   std::cout << "IArray2D::WriteArray: ";
489:   unsigned int line_number = 0;
490:
491:   try {
492:     for (const auto &i : indices(F)) {
493:       for (const auto &j : indices(F[i])) {
494:         // Write to file
495:         file << std::setprecision(15) << F[i][j][0] << "," << F[i][j][1] << "," << F[i][j][2] << ","
496:              << F[i][j][3] << std::endl;
497:
498:         ++line_number;
```

```
499:       }
500:     }
501:   } catch (...) {
502:     throw std::invalid_argument("IArray2D:WriteArray: Error in file " + filename + " at line " +
503:                                 std::to_string(line_number));
504:   }
505:
506:   file.close();
507:   return true;
508: }
509:
510: // Read the array from a file
511: bool IArray2D::ReadArray(const std::string &filename) {
512:   std::ifstream file;
513:   file.open(filename);
514:   if (!file.is_open()) {
515:     std::string str = "IArray2D::ReadArray: Fatal IO-error with: " + filename;
516:     return false;
517:   }
518:
519:   std::string  line;
520:   unsigned int fills       = 0;
521:   unsigned int line_number = 0;
522:   std::cout << "IArray2D::ReadArray: ";
523:
524:   try {
525:     for (const auto &i : indices(F)) {
526:       for (const auto &j : indices(F[i])) {
527:         // Read every line from the stream
528:         getline(file, line);
529:
530:         std::istringstream       stream(line);
531:         std::vector<std::string> columns;
532:         std::string              element;
533:
534:         // Get every line element (4 of them) separated by separator
535:         int k = 0;
536:         while (getline(stream, element, ',')) {
537:           F[i][j][k] = std::stod(element);  // string to double
538:           ++k;
539:           ++fills;
540:         }
541:         ++line_number;
542:       }
543:     }
544:
545:   } catch (...) {
546:     throw std::invalid_argument("IArray2D:ReadArray: Error in file " + filename + " at line " +
547:                                 std::to_string(line_number));
548:   }
549:
550:   file.close();
551:
552:   if (fills != 4 * (N[0] + 1) * (N[1] + 1)) {
553:     std::string str = "Corrupted file: " + filename;
554:     std::cout << str << std::endl;
555:     return false;
556:   }
557:   std::cout << rang::fg::green << "[DONE]" << rang::fg::reset << std::endl;
558:   return true;
559: }
560:
561: // Standard 2D-bilinear interpolation f(a,b) = Z, and derivative dZ
562: //
563: //
564: // [REFERENCE: en.wikipedia.org/wiki/Bilinear_interpolation]
565: std::pair<double, double> IArray2D::Interpolate2D(double a, double b) const {
566:   const double EPS = 1e-5;
567:
568:   if (a < MIN[0]) { a = MIN[0]; }  // Truncate before (possible) logarithm
569:   if (b < MIN[1]) { b = MIN[1]; }  // Truncate before (possible) logarithm
570:
571:   // Logarithmic stepping or not
572:   if (islog[0]) { a = std::log(a); }
573:   if (islog[1]) { b = std::log(b); }
574:
575:   if (a > MAX[0] * (1 + EPS) || b > MAX[1] * (1 + EPS)) {
576:     printf(
577:         "Interpolate2D(%s,%s) Input out of grid domain: "
578:         "%s = %0.3f [%0.3f, %0.3f], %s = %0.3f [%0.3f, %0.3f] \n",
579:         name[0].c_str(), name[1].c_str(), name[0].c_str(), a, MIN[0], MAX[0], name[1].c_str(), b,
580:         MIN[1], MAX[1]);
581:   }
```



```
582:
583:   // Get indices
584:   int i = std::floor((a - MIN[0]) / STEP[0]);
585:   int j = std::floor((b - MIN[1]) / STEP[1]);
586:
587:   // Lower boundary protection
588:   if (i < 0) { i = 0; }  // Int needed for this (not unsigned int)
589:   if (j < 0) { j = 0; }
590:
591:   // Upper boundary protection
592:   if (i >= (int)N[0]) { i = N[0] - 1; }  // We got N+1 elements in F
593:   if (j >= (int)N[1]) { j = N[1] - 1; }
594:
595:   // Aux variables for readability
596:   const unsigned int X = 0;
597:   const unsigned int Y = 1;
598:
599:   const double x1 = F[i][j][X];
600:   const double x2 = F[i + 1][j][X];
601:   const double y1 = F[i][j][Y];
602:   const double y2 = F[i][j + 1][Y];
603:
604:   const double xstep = STEP[0];
605:   const double ystep = STEP[1];
606:   const double xval  = a;
607:   const double yval  = b;
608:
609:   double values[2] = {0.0};
610:
611:   // 2 == Z, 3 == dZ
612:   for (std::size_t C = 2; C <= 3; ++C) {
613:     const double Q11 = F[i][j][C];
614:     const double Q12 = F[i][j + 1][C];
615:     const double Q21 = F[i + 1][j][C];
616:     const double Q22 = F[i + 1][j + 1][C];
617:
618:     // Interpolated valued
619:     values[C - 2] =
620:         ((y2 - yval) / ystep) * ((x2 - xval) / xstep * Q11 + (xval - x1) / xstep * Q21) +
621:         ((yval - y1) / ystep) * ((x2 - xval) / xstep * Q12 + (xval - x1) / xstep * Q22);
622:   }
623:
624:   return {values[0], values[1]};
625: }
626:
627: }  // namespace gra
628:
```



```
1: // Form factors, structure functions, Regge trajectories etc. parametrizations
2: //
3: // (c) 2017-2020 Mikael Mieskolainen
4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
5:
6: // C++
7: #include <complex>
8: #include <cstdlib>
9: #include <iostream>
10: #include <map>
11: #include <vector>
12:
13: // Own
14: #include "Granititi/MForm.h"
15: #include "Granititi/MGranititi.h"
16: #include "Granititi/MKinematics.h"
17: #include "Granititi/MMath.h"
18: #include "Granititi/MProcess.h"
19: #include "Granititi/MQED.h"
20: #include "Granititi/MSpin.h"
21:
22: // Libraries
23: #include "json.hpp"
24: #include "rang.hpp"
25:
26:
27: using gra::math::PI;
28: using gra::math::abs2;
29: using gra::math::msqrt;
30: using gra::math::pow2;
31: using gra::math::pow3;
32: using gra::math::zi;
33:
34: using gra::PDG::mp;
35: using gra::PDG::mpi;
36:
37: namespace gra {
38:
39: // Structure functions and nuclear form factors
40: namespace PARAM_STRUCTURE {
41:
42: std::string F2        = "CKMT";
43: std::string EM        = "DIPOLE";
44: std::string QED_alpha = "ZERO";
45:
46: bool initialized = false;
47: // Read parameters from file
48: void ReadParameters(const std::string &modelfile) {
49:   try {
50:     const std::string data = gra::aux::GetInputData(modelfile);
51:     nlohmann::json    j    = nlohmann::json::parse(data);
52:
53:     const std::string XID = "PARAM_STRUCTURE";
54:
55:     PARAM_STRUCTURE::F2        = j.at(XID).at("F2");
56:     PARAM_STRUCTURE::EM        = j.at(XID).at("EM");
57:     PARAM_STRUCTURE::QED_alpha = j.at(XID).at("QED_alpha");
58:
59:     initialized = true;
60:
61:   } catch (...) {
62:     std::string str = "PARAM_STRUCTURE::ReadParameters: Error parsing " + modelfile +
63:                       "  (Check for extra/missing commas)";
64:     throw std::invalid_argument(str);
65:   }
66: }
67:
68: }  // namespace PARAM_STRUCTURE
69:
70: // Model parameters
71: namespace PARAM_SOFT {
72: // Pomeron trajectory
73: double DELTA_P = 0.0;
74: double ALPHA_P = 0.0;
75:
76: // Couplings
77: double gN_P  = 0.0;
78: double gN_O  = 0.0;
79: double g3P   = 0.0;
80: double gamma = 0.0;
81:
82: // Proton form factor
83: double fc1 = 0.0;
```



```
84: double fc2 = 0.0;
85:
86: // Pion loop
87: double fc3 = 0.0;
88:
89: // On/Off
90: bool ODDERON_ON = false;
91:
92: bool initialized = false;
93:
94: std::string GetHashString() {
95:   std::string str = std::to_string(PARAM_SOFT::DELTA_P) + std::to_string(PARAM_SOFT::ALPHA_P) +
96:                     std::to_string(PARAM_SOFT::gN_P) + std::to_string(PARAM_SOFT::gN_O) +
97:                     std::to_string(PARAM_SOFT::fc1) + std::to_string(PARAM_SOFT::fc2) +
98:                     std::to_string(PARAM_SOFT::fc3) + std::to_string(PARAM_SOFT::ODDERON_ON);
99:   return str;
100: }
101:
102: void PrintParam() {
103:   printf("PARAM_SOFT:: Soft model parameters: \n\n");
104:   printf("- DELTA_P    = %0.5f \n", DELTA_P);
105:   printf("- ALPHA_P    = %0.5f [GeV^{-2}] \n", ALPHA_P);
106:   printf("- gN_P       = %0.5f [GeV^{-1}] \n", gN_P);
107:   printf("- gN_O       = %0.5f [GeV^{-1}] \n", gN_O);
108:   printf("- g3P        = %0.5f \n", g3P / gN_P);  // Convention
109:   printf("- gamma      = %0.5f \n", gamma);
110:   printf("- fc1        = %0.5f [GeV^2] \n", fc1);
111:   printf("- fc2        = %0.5f [GeV^2] \n", fc2);
112:   printf("- fc3        = %0.5f [GeV^2] \n", fc3);
113:   printf("- ODDERON_ON = ");
114:   std::cout << (ODDERON_ON ? "true" : "false") << std::endl;
115:   std::cout << std::endl << std::endl;
116: }
117:
118: // Read parameters from file
119: void ReadParameters(const std::string &modelfile) {
120:   try {
121:     const std::string data = gra::aux::GetInputData(modelfile);
122:     nlohmann::json    j    = nlohmann::json::parse(data);
123:
124:     const std::string XID = "PARAM_SOFT";
125:
126:     PARAM_SOFT::DELTA_P = j.at(XID).at("DELTA_P");
127:     PARAM_SOFT::ALPHA_P = j.at(XID).at("ALPHA_P");
128:     PARAM_SOFT::gN_P    = j.at(XID).at("gN_P");
129:     PARAM_SOFT::gN_O    = j.at(XID).at("gN_O");
130:
131:     double triple3P   = j.at(XID).at("g3P");
132:     PARAM_SOFT::g3P   = triple3P * PARAM_SOFT::gN_P;  // Convention
133:     PARAM_SOFT::gamma = j.at(XID).at("gamma");
134:
135:     PARAM_SOFT::fc1 = j.at(XID).at("fc1");
136:     PARAM_SOFT::fc2 = j.at(XID).at("fc2");
137:     PARAM_SOFT::fc3 = j.at(XID).at("fc3");
138:
139:     PARAM_SOFT::ODDERON_ON = j.at(XID).at("ODDERON_ON");
140:
141:     initialized = true;
142:     PARAM_SOFT::PrintParam();
143:
144:   } catch (...) {
145:     std::string str = "PARAM_SOFT::ReadParameters: Error parsing " + modelfile +
146:                       "  (Check for extra/missing commas)";
147:     throw std::invalid_argument(str);
148:   }
149: }
150:
151: }  // namespace PARAM_SOFT
152:
153: // Flat amplitude parameters
154: namespace PARAM_FLAT {
155: int    active = 0;
156: double b      = 0.0;
157:
158: bool initialized = false;
159:
160: // Read parameters from file
161: void ReadParameters(const std::string &modelfile) {
162:   try {
163:     const std::string data = gra::aux::GetInputData(modelfile);
164:     nlohmann::json    j    = nlohmann::json::parse(data);
165:
166:     const std::string XID = "PARAM_FLAT";
```

```
167:
168:     PARAM_FLAT::b = j.at(XID).at("b");
169:
170:     initialized = true;
171:   } catch (...) {
172:     std::string str = "PARAM_FLAT::ReadParameters: Error parsing " + modelfile +
173:                       " (Check for extra/missing commas)";
174:     throw std::invalid_argument(str);
175:   }
176: }
177: }  // namespace PARAM_FLAT
178:
179: // Forward proton excitation
180: namespace PARAM_NSTAR {
181: std::string          fragment = "none";
182: std::vector<double> rc       = {0.0, 0.0, 0.0};
183:
184: bool initialized = false;
185:
186: // Read parameters from file
187: void ReadParameters(const std::string &modelfile) {
188:   try {
189:     const std::string data = gra::aux::GetInputData(modelfile);
190:     nlohmann::json    j    = nlohmann::json::parse(data);
191:
192:     const std::string XID = "PARAM_NSTAR";
193:
194:     PARAM_NSTAR::fragment  = j.at(XID).at("fragment");
195:     std::vector<double> rc = j.at(XID).at("rc");
196:     PARAM_NSTAR::rc        = rc;
197:
198:     // Make sure they sum to one
199:     const double sum_rc = std::accumulate(rc.begin(), rc.end(), 0);
200:     for (std::size_t i = 0; i < PARAM_NSTAR::rc.size(); ++i) { PARAM_NSTAR::rc[i] /= sum_rc; }
201:
202:     initialized = true;
203:   } catch (...) {
204:     std::string str = "PARAM_NSTAR::ReadParameters: Error parsing " + modelfile +
205:                       " (Check for extra/missing commas)";
206:     throw std::invalid_argument(str);
207:   }
208: }
209: }  // namespace PARAM_NSTAR
210:
211: namespace form {
212:
213: // Read resonance parameters
214: gra::PARAM_RES ReadResonance(const std::string &resparam_str, MRandom &rng) {
215:   // ========================================================================
216:   // Find global resonance parameters
217:
218:   bool        SPINGEN = false;
219:   bool        SPINDEC = false;
220:   std::string FRAME   = "null";
221:   int         JMAX    = 0;
222:
223:   {
224:     // Read and parse
225:     const std::string fullpath =
226:         gra::aux::GetBasePath(2) + "/modeldata/" + gra::MODELPARAM + "/GENERAL.json";
227:
228:     const std::string data = gra::aux::GetInputData(fullpath);
229:     nlohmann::json    j;
230:
231:     try {
232:       j = nlohmann::json::parse(data);
233:     } catch (...) {
234:       std::string str =
235:           "form::ReadResonance: Error parsing " + fullpath + " (Check for extra/missing commas)";
236:       throw std::invalid_argument(str);
237:     }
238:
239:     SPINGEN = j.at("PARAM_SPIN").at("SPINGEN");
240:     SPINDEC = j.at("PARAM_SPIN").at("SPINDEC");
241:     FRAME   = j.at("PARAM_SPIN").at("FRAME");
242:     JMAX    = j.at("PARAM_SPIN").at("JMAX");
243:   }
244:
245:   // ========================================================================
246:
247:   std::cout << "gra::form::ReadResonance: Reading " + resparam_str + " ";
248:
249:   // Create a JSON object from file
```



```
250:   std::string inputfile =
251:       gra::aux::GetBasePath(2) + "/modeldata/" + gra::MODELPARAM + "/" + resparam_str;
252:
253:   // Read and parse
254:   std::string    data;
255:   nlohmann::json j;
256:
257:   try {
258:     data = gra::aux::GetInputData(inputfile);
259:     j    = nlohmann::json::parse(data);
260:   } catch (...) {
261:     throw std::invalid_argument("form::ReadResonance: Error parsing '" + resparam_str + "'");
262:   }
263:
264:   // Resonance parameters
265:   gra::PARAM_RES res;
266:
267:   try {
268:     // ------------------------------------------------------------------
269:     // Collect global variables
270:     res.SPINGEN = SPINGEN;
271:     res.SPINDEC = SPINDEC;
272:     res.FRAME   = FRAME;
273:     res.JMAX    = JMAX;
274:     // ------------------------------------------------------------------
275:
276:     // Complex coupling
277:     double g_A   = j.at("PARAM_RES").at("g_A");
278:     double g_phi = j.at("PARAM_RES").at("g_phi");
279:     res.g        = g_A * std::exp(std::complex<double>(0, 1) * g_phi);
280:
281:     // Form factor parameter
282:     res.g_FF = j.at("PARAM_RES").at("g_FF");
283:
284:     // PDG code
285:     res.p.pdg = j.at("PARAM_RES").at("PDG");
286:
287:     // Mass
288:     res.p.mass = j.at("PARAM_RES").at("M");
289:     if (res.p.mass < 0) {
290:       std::string str = "MAux:ReadResonance:: <" + resparam_str + "> Invalid M < 0 !";
291:       throw std::invalid_argument(str);
292:     }
293:
294:     // Width
295:     res.p.width = j.at("PARAM_RES").at("W");
296:     if (res.p.width < 0) {
297:       std::string str = "MAux:ReadResonance:: <" + resparam_str + "> Invalid W < 0 !";
298:       throw std::invalid_argument(str);
299:     }
300:
301:     // Spin
302:     const double J = j.at("PARAM_RES").at("J");
303:     res.p.spinX2   = J * 2;
304:     if (res.p.spinX2 < 0) {
305:       std::string str = "MAux:ReadResonance:: <" + resparam_str + "> Invalid J < 0 !";
306:       throw std::invalid_argument(str);
307:     }
308:
309:     // Parity
310:     res.p.P = j.at("PARAM_RES").at("P");
311:     if (!(res.p.P == -1 || res.p.P == 1)) {
312:       std::string str = "MAux:ReadResonance:: <" + resparam_str + "> Invalid P (not -1 or 1) !";
313:       throw std::invalid_argument(str);
314:     }
315:
316:     // ------------------------------------------------------------------
317:     // Tensor Pomeron couplings vector
318:     std::vector<double> g_Tensor = j.at("PARAM_RES").at("g_Tensor");
319:     res.g_Tensor                 = g_Tensor;
320:
321:     if (J == 0 && res.g_Tensor.size() != 2) {
322:       throw std::invalid_argument("MAux::ReadResonance:: <" + resparam_str +
323:                                   "> Tensor Pomeron coupling array should be of size 2 for J = 0");
324:     }
325:     if (J == 1 && res.g_Tensor.size() != 2) {
326:       throw std::invalid_argument("MAux::ReadResonance:: <" + resparam_str +
327:                                   "> Tensor Pomeron coupling array should be of size 2 for J = 1");
328:     }
329:     if (J == 2 && res.g_Tensor.size() != 7) {
330:       throw std::invalid_argument("MAux::ReadResonance:: <" + resparam_str +
331:                                   "> Tensor Pomeron coupling array should be of size 7 for J = 2");
332:     }
```



```
333:     // ------------------------------------------------------------------
334:
335:     // Validity of these is taken care of in the functions
336:     res.BW = j.at("PARAM_RES").at("BW");
337:
338:     const bool P_conservation = true;
339:
340:     const int                  n = res.p.spinX2 + 1;  // n = 2J + 1
341:     MMatrix<std::complex<double>> rho(n, n);
342:
343:     // If we have spin
344:     if (res.p.spinX2 != 0) {
345:       // Draw random density matrices (until the number set by user)
346:       if (j.at("PARAM_RES").at("random_rho") > 0) {
347:         for (std::size_t k = 0; k < j.at("PARAM_RES").at("random_rho"); ++k) {
348:           rho = gra::spin::RandomRho(res.p.spinX2 / 2.0, P_conservation, rng);
349:         }
350:
351:         // Construct spin density matrix from the input
352:       } else {
353:         for (std::size_t a = 0; a < rho.size_row(); ++a) {
354:           for (std::size_t b = 0; b < rho.size_col(); ++b) {
355:             const double Re = j.at("PARAM_RES").at("rho_real").at(a).at(b);
356:             const double Im = j.at("PARAM_RES").at("rho_imag").at(a).at(b);
357:             rho[a][b]       = Re + zi * Im;
358:           }
359:         }
360:         // Check positivity conditions
361:         if (gra::spin::Positivity(rho, res.p.spinX2 / 2.0) == false) {
362:           std::string str = "gra::form::ReadResonance: <" + resparam_str +
363:                             "> Input density matrix not positive definite!";
364:           throw std::invalid_argument(str);
365:         }
366:       }
367:       res.rho = rho;
368:     }
369:     std::cout << rang::fg::green << "[DONE]" << rang::fg::reset << std::endl;
370:   } catch (nlohmann::json::exception &e) {
371:     throw std::invalid_argument("form::ReadResonance: Missing parameter in '" + resparam_str +
372:                                 "' : " + e.what());
373:   }
374:
375:   return res;
376: }
377:
378: // Regge signature factor, alpha_t is alpha(t), and signature sigma = +-1
379: //
380: // Remember that denominator hits pole at every integer PI * alpha(t)
381: //
382: std::complex<double> ReggeEta(double alpha_t, double sigma) {
383:   const double denom = std::sin(PI * alpha_t);
384:   return -(1.0 + sigma * std::exp(-zi * PI * alpha_t)) / denom;
385: }
386:
387: // Regge signature factor for linear trajectories at t -> 0
388: //
389: //         t = Mandelstam t
390: //  alpha_t0 = alpha(0)
391: //        ap = alpha'
392: //     sigma = +-1
393: //
394: std::complex<double> ReggeEtaLinear(double t, double alpha_t0, double ap, double sigma) {
395:   return ReggeEta(alpha_t0, sigma) * std::exp(-zi * PI / 2.0 * ap * t);
396: }
397:
398: // ------------------------------------------------------------------------
399: // Non-linear Pomeron trajectory functional form:
400: //
401: // [REFERENCE: Khoze, Martin, Ryskin, arxiv.org/abs/hep-ph/0007359]
402: double S3PomAlpha(double t) {
403:   // Additive quark model Beta_pi / Beta_p = 2/3 ->
404:   const double BETApi = (2.0 / 3.0) * PARAM_SOFT::gN_F;
405:   const double PIC    = (pow2(BETApi) * pow2(mpi)) / (32.0 * pow3(PI));
406:
407:   // Pomeron trajectory value
408:   const double h       = PIC * S3HPL(4.0 * pow2(mpi) / std::abs(t),
409:                                      t);  // pion loop insert (ADD with minus sign)
410:   const double alpha_P = 1.0 + PARAM_SOFT::DELTA_P + PARAM_SOFT::ALPHA_P * t - h;
411:
412:   return alpha_P;
413: }
414:
415: // Pion loop insert to the non-linear Pomeron trajectory
```



```
416: double S3HPL(double tau, double t) {
417:   const double m     = 1.0;  // fixed scale (GeV)
418:   const double sqrtau = msqrt(1 + tau);
419:
420:   // Pion-Pomeron form factor parametrization
421:   const double F_pi = 1.0 / (1.0 - t / PARAM_SOFT::fc3);
422:
423:   return (4.0 / tau) * pow2(F_pi) *
424:          (2.0 * tau - std::pow(1.0 + tau, 3.0 / 2.0) * std::log((sqrtau + 1.0) / (sqrtau - 1.0)) +
425:           std::log((m * m) / (mpi * mpi)));
426: }
427: // ------------------------------------------------------------------------
428:
429: // Elastic proton form factor parametrization
430: //
431: // <apply at amplitude level>
432: // [REFERENCE: Khoze, Martin, Ryskin, arxiv.org/abs/hep-ph/0007359]
433: double S3F(double t) {
434:   return (1.0 / (1 - t / PARAM_SOFT::fc1)) * (1.0 / (1 - t / PARAM_SOFT::fc2));
435: }
436:
437: // Proton inelastic form factor / structure function
438: // parametrization for Pomeron processes [THIS FUNCTION IS ANSATZ - IMPROVE!]
439: //
440: // Motivated by arxiv.org/abs/hep-ph/9305319
441: //
442: // <apply at amplitude level>
443: double S3FINEL(double t, double M2) {
444:   constexpr double DELTA_P = 0.0808;
445:   constexpr double a       = 0.5616;  // GeV^{2}
446:   double           f       = std::pow(std::abs(t) / (M2 * (std::abs(t) + a)), 0.5 * (1 + DELTA_P));
447:
448:   return f;
449: }
450:
451: // Proton inelastic structure function F2(x,Q^2) parametrization
452: //
453: // The basic idea is that at low-Q^2, a fully non-perturbative description
454: // (parametrization) is needed.
455: // At high Q^2, DGLAP evolution could be done in log(Q^2) starting from the
456: // input description.
457: //
458: // Now, some (very) classic ones have been implemented. Add new one here!
459: //
460: // [REFERENCE: Donnachie, Landshoff, arxiv.org/abs/hep-ph/9305319]
461: // [REFERENCE: Capella, Kaidalov, Merino, Tran Tranh Van, arxiv.org/abs/hep-ph/9405338v1]
462: //
463: double F2xQ2(double xbj, double Q2) {
464:   if (PARAM_STRUCTURE::F2 == "DL") {
465:     constexpr double A = 0.324;
466:     constexpr double B = 0.098;
467:
468:     constexpr double DELTA_P = 0.0808;
469:     constexpr double DELTA_R = 0.5475;
470:
471:     constexpr double a = 0.5619916928786383;
472:     constexpr double b = 0.011133;
473:
474:     const double F2 = A * std::pow(xbj, -DELTA_P) * std::pow(Q2 / (Q2 + a), 1 + DELTA_P) +
475:                       B * std::pow(xbj, 1 - DELTA_R) * std::pow(Q2 / (Q2 + b), DELTA_R);
476:
477:     return F2;
478:   }
479:
480:   else if (PARAM_STRUCTURE::F2 == "CKMT") {
481:     constexpr double A       = 0.1502;
482:     constexpr double B_u     = 1.2064;
483:     constexpr double B_d     = 0.1798;
484:     constexpr double alpha_R = 0.4150;
485:     constexpr double DELTA_0 = 0.0800;
486:
487:     constexpr double a = 0.2631;
488:     constexpr double b = 0.6452;
489:     constexpr double c = 3.5489;
490:     constexpr double d = 1.1170;
491:
492:     const double n_Q2     = (3.0 / 2.0) * (1 + Q2 / (Q2 + c));
493:     const double DELTA_Q2 = DELTA_0 * (1 + (2 * Q2) / (Q2 + d));
494:
495:     const double C1 = std::pow(Q2 / (Q2 + a), 1.0 + DELTA_Q2);
496:     const double C2 = std::pow(Q2 / (Q2 + b), alpha_R);
497:
498:     const double F2 = A * std::pow(xbj, -DELTA_Q2) * std::pow(1 - xbj, n_Q2 + 4.0) * C1 +
```

```
499:                          std::pow(xbj, 1.0 - alpha_R) *
500:                               (B_u * std::pow(1 - xbj, n_Q2) + B_d * std::pow(1 - xbj, n_Q2 + 1.0)) *
501:                               C2;
502:
503:     return F2;
504:   } else {
505:     throw std::invalid_argument("gra::form::F2xQ2: Unknown PARAM_STRUCTURE::F2 = " +
506:                                 PARAM_STRUCTURE::F2);
507:   }
508: }
509:
510: // "Purely magnetic structure function"
511: //
512: // Callan-Gross relation for spin-1/2: F_2(x) = 2xF_1(x) under Bjorken scaling
513: // For spin-0, F_1(x) = 0
514: //
515: // Longitudinal structure function definition (e.g. QCD):
516: // F_L(xbj,Q2) = (1 + 4*pow2(xbj*mp)/Q2) * F2(xbj, Q2) - 2xbj * F1(xbj,Q2)
517: //
518: double F1xQ2(double xbj, double Q2) { return F2xQ2(xbj, Q2) / (2.0 * xbj); }
519:
520: // QED: Proton magnetic moment in magneton units (mu_p / mu_N)
521: double mu_ratio() { return 2.792847337; }
522:
523: // ============================================================================
524: // Photon flux densities and form factors, input Q^2 as positive
525:
526: // kT unintegrated coherent EPA photon flux as in:
527: //
528: // [REFERENCE: Luszczak, Schaefer, Szczurek, arxiv.org/abs/1802.03244]
529: //
530: // Form factors:
531: // [REFERENCE, Punjabi et al., arxiv.org/abs/1503.01452v4]
532: //
533: // Proton electromagnetic form factors: Basic notions, present
534: // achievements and future perspectives, Physics Reports, 2015
535: // <www.sciencedirect.com/science/article/pii/S0370157314003184>
536: //
537: // [REFERENCE: Budnev, Ginzburg, Meledin, Serbo, EPA paper, Physics Reports, 1976]
538: // <www.sciencedirect.com/science/article/pii/0370157375900095>
539: //
540: //
541: // Proton EM form factor F1 (Dirac)
542: double F1(double Q2) {
543:   Q2                = std::abs(Q2);
544:   const double tau = Q2 / pow2(2 * mp);
545:
546:   return 1.0 / (tau + 1) * G_E(Q2) + tau / (tau + 1) * G_M(Q2);
547: }
548:
549: // Proton EM form factor F2 (Pauli)
550: double F2(double Q2) {
551:   Q2                = std::abs(Q2);
552:   const double tau = Q2 / pow2(2 * mp);
553:
554:   return -1.0 / (tau + 1) * G_E(Q2) + 1.0 / (tau + 1) * G_M(Q2);
555: }
556:
557: // Rosenbluth separation:
558: // low-Q^2 dominated by G_E, high-Q^2 dominated by G_M
559:
560: // "Sachs Form Factor" goes as follows:
561: // G_E(0) = 1 for proton, 0 for neutron
562: // G_M(0) = mu_p for proton, mu_n for neutrons
563:
564: // <http://www.scholarpedia.org/article/Nucleon_Form_factors>
565:
566: double G_E(double Q2) {
567:   Q2 = std::abs(Q2);  // For safety
568:
569:   if (PARAM_STRUCTURE::EM == "DIPOLE") {
570:     return G_E_DIPOLE(Q2);
571:   } else if (PARAM_STRUCTURE::EM == "KELLY") {
572:     return G_E_KELLY(Q2);
573:   } else {
574:     throw std::invalid_argument("gra::form::G_E: Unknown proton EM-form factor chosen = " +
575:                                 PARAM_STRUCTURE::EM);
576:   }
577: }
578:
579: double G_M(double Q2) {
580:   Q2 = std::abs(Q2);  // For safety
581:
```



```
582:   if (PARAM_STRUCTURE::EM == "DIPOLE") {
583:     return G_M_DIPOLE(Q2);
584:   } else if (PARAM_STRUCTURE::EM == "KELLY") {
585:     return G_M_KELLY(Q2);
586:   } else {
587:     throw std::invalid_argument("gra::form::G_M: Unknown proton EM-form factor chosen = " +
588:                                 PARAM_STRUCTURE::EM);
589:   }
590: }
591:
592:
593: // The simplest possible: Dipole parametrization of nucleon EM-form factors
594: //
595: //
596: double G_E_DIPOLE(double Q2) {
597:   return G_M(Q2) / mu_ratio();  // Scaling assumption
598: }
599: double G_M_DIPOLE(double Q2) {
600:   constexpr double lambda2 = 0.71;  // Dipole parameter GeV^2
601:   return mu_ratio() / pow2(1.0 + Q2 / lambda2);
602: }
603:
604: // Simple parametrization of nucleon EM-form factors
605: //
606: // [REFERENCE: Kelly, journals.aps.org/prc/pdf/10.1103/PhysRevC.70.068202]
607: double G_E_KELLY(double Q2) {
608:   static const std::vector<double> a = {1, -0.24};
609:   static const std::vector<double> b = {10.98, 12.82, 21.97};
610:
611:   const double tau = Q2 / pow2(2 * mp);
612:
613:   // Numerator
614:   double num = 0.0;   // 0
615:   num += a[0];        // a_0 tau^0
616:   num += a[1] * tau;  // a_1 tau^1
617:
618:   // Denominator
619:   double den = 1.0;         // 1.0
620:   den += b[0] * tau;        // b_1 tau^1
621:   den += b[1] * pow2(tau);  // b_2 tau^2
622:   den += b[2] * pow3(tau);  // b_3 tau^3
623:
624:   return num / den;
625: }
626:
627: double G_M_KELLY(double Q2) {
628:   static const std::vector<double> a = {1, 0.12};
629:   static const std::vector<double> b = {10.97, 18.86, 6.55};
630:
631:   const double tau = Q2 / pow2(2 * mp);
632:
633:   // Numerator
634:   double num = 0.0;   // 0
635:   num += a[0];        // a_0 tau^0
636:   num += a[1] * tau;  // a_1 tau^1
637:
638:   // Denominator
639:   double den = 1.0;         // 1.0
640:   den += b[0] * tau;        // b_1 tau^1
641:   den += b[1] * pow2(tau);  // b_2 tau^2
642:   den += b[2] * pow3(tau);  // b_3 tau^3
643:
644:   return mu_ratio() * num / den;
645: }
646:
647: // Coherent photon flux from proton
648: // xi ~ longitudinal momentum loss [0,1]
649: // t  ~ Mandelstam t
650: // pt ~ proton transverse momentum
651: //
652: //
653: //  p ---------F_E---------> p' with xi = 1 - p'^_z'/p_z
654: //                  $
655: //                    $
656: //                      $
657: //
658: // Factors applied here, compatible with 2 -> N phase space sampling:
659: //
660: //  1/xi     [~ sub Moller flux]
661: //  1/pt2    [~ kt-factorization] (cancels with pt2 from numerator)
662: //  16pi^2   [~ kinematics volume factor]
663: //
664: double CohFlux(double xi, double t, double pt) {
```



```
665:   const double pt2 = pow2(pt);
666:   const double xi2 = pow2(xi);
667:   const double mp2 = pow2(mp);
668:   const double Q2  = std::abs(t);
669:
670:   const double PART1 =
671:       (4.0 * pow2(mp) * pow2(G_E(Q2)) + Q2 * pow2(G_M(Q2))) / (4.0 * pow2(mp) + Q2);
672:   const double PART2 = pow2(G_M(Q2));
673:   const double DELTA = pt2 / (pt2 + xi2 * mp2);
674:
675:   double f =
676:       qed::alpha_QED() / PI * ((1.0 - xi) * pow2(DELTA) * PART1 + (xi2 / 4.0) * DELTA * PART2);
677:
678:   // Factors
679:   f /= xi;
680:   f /= pt2;
681:   f *= 16.0 * gra::math::PIPI;
682:
683:   return f;  // Use at cross section level
684: }
685:
686: // Incoherent photon flux from a dissociated proton with mass M.
687: // When M -> mp, this reproduces CohFlux() if
688: // F2(x,Q^2) does reproduce the elastic limit (not all parametrizations do)
689: //
690: //
691: //  p ---------F2(x,Q^2)----->  p* with xi = 1 - p'^_z / p_z
692: //                  ------------->
693: //                  ---x---------->
694: //                     $
695: //                       $
696: //                         $
697: //
698: // Factors applied as with CohFlux() above.
699: //
700: double IncohFlux(double xi, double t, double pt, double M2) {
701:   constexpr double mp2 = pow2(mp);
702:
703:   const double pt2   = pow2(pt);
704:   const double xi2   = pow2(xi);
705:   const double Q2    = std::abs(t);
706:   const double xbj   = Q2 / (Q2 + M2 - mp2);  // Bjorken-x
707:   const double DELTA = pt2 / (pt2 + xi * (M2 - mp2) + xi2 * mp2);
708:
709:   double f = qed::alpha_QED() / PI *
710:              ((1.0 - xi) * pow2(DELTA) * F2xQ2(xbj, Q2) / (Q2 + M2 - mp2) +
711:               (xi2 / (4.0 * pow2(xbj))) * DELTA * 2.0 * xbj * F1xQ2(xbj, Q2) / (Q2 + M2 - mp2));
712:
713:   // Factors
714:   f /= xi;
715:   f /= pt2;
716:   f *= 16.0 * gra::math::PIPI;
717:
718:   return f;  // Use at cross section level
719: }
720:
721: // Drees-Zeppenfeld proton coherent gamma flux (collinear)
722: double DZFlux(double x) {
723:   const double Q2min = (pow2(mp) * pow2(x)) / (1.0 - x);
724:   const double A     = 1.0 + 0.71 / Q2min;
725:
726:   double f = qed::alpha_QED() / (2.0 * PI * x) * (1.0 + pow2(1.0 - x)) *
727:              (std::log(A) - 11.0 / 6.0 + 3.0 / A - 3.0 / (2.0 * pow2(A)) + 1.0 / (3 * pow3(A)));
728:
729:   return f;  // Use at cross section level
730: }
731:
732: // Breit-Wigner propagators / form factors
733: //
734: //
735: //
736: // Useful identity for normalization:
737: //
738: // \int dm^2 \frac{1}{(m^2 - M0^2)^2 + M0^2Gamma^2} \equiv \frac{\pi}{M0 Gamma}
739: //
740: // based on squaring the complex propagator
741: // D(m^2) = 1 / (m^2 - M0^2 + iM0*Gamma), and integrating.
742: //
743: std::complex<double> CBW(const gra::LORENTZSCALAR &lts, const gra::PARAM_RES &resonance) {
744:   switch (resonance.BW) {
745:     case 1:
746:       return CBW_FW(lts.m2, resonance.p.mass, resonance.p.width);
747:     case 2:
```



```
748:       return CBW_RW(lts.m2, resonance.p.mass, resonance.p.width);
749:     case 3:
750:       return CBW_BF(lts.m2, resonance.p.mass, resonance.p.width, resonance.p.spinX2 / 2.0,
751:                     lts.decaytree[0].p4.M(), lts.decaytree[1].p4.M());
752:     case 4:
753:       return CBW_JR(lts.m2, resonance.p.mass, resonance.p.width, resonance.p.spinX2 / 2.0);
754:
755:     default:
756:       throw std::invalid_argument("CBW: Unknown BW (Breit-Wigner) parameter: " +
757:                                   std::to_string(resonance.BW));
758:   }
759: }
760:
761: // See e.g.
762: // [REFERENCE: TASI Lectures on propagators, users.ictp.it/~smr2244/tait-supplemental.pdf]
763: // [REFERENCE: Cacciapaglia, Deandrea, Curtis, arxiv.org/abs/0906.3417v2]
764: // [REFERENCE:
765: // www.t2.ucsd.edu/twiki2/pub/UCSDTier2/Physics2145Spring2015/ajw-breit-wigner-cbx99-55.pdf]
766:
767: // ----------------------------------------------------------------------
768: // Delta function \delta(\hat{s} - M_0^2) replacement function:
769: //    \int d\hat{s} \delta(\hat{s} - M_0^2) -> \int d\hat{s}
770: //    deltaBW(\hat{s},M0,Gamma)
771: //
772: // To be applied at cross section level
773: double deltaBWxsec(double shat, double M0, double Gamma) {
774:   return M0 * Gamma / PI / (pow2(shat - pow2(M0)) + pow2(M0 * Gamma));
775: }
776:
777: // To be applied at amplitude level
778: double deltaBWamp(double shat, double M0, double Gamma) {
779:   return msqrt(deltaBWxsec(shat, M0, Gamma));
780: }
781: // ----------------------------------------------------------------------
782:
783: // ----------------------------------------------------------------------
784: // Breit-Wigner propagator parametrizations
785:
786: // 1. Complex Fixed Width Relativistic Breit-Wigner
787: std::complex<double> CBW_FW(double m2, double M0, double Gamma) {
788:   return -1.0 / (m2 - M0 * M0 + zi * M0 * Gamma);
789: }
790:
791: // 2. Complex Running Width Relativistic Breit-Wigner
792: std::complex<double> CBW_RW(double m2, double M0, double Gamma) {
793:   return -1.0 / (m2 - M0 * M0 + zi * std::sqrt(m2) * Gamma);
794: }
795:
796: // 3. J = 0,1,2 Complex Relativistic Breit-Wigner
797: // with angular barrier effects type of Bleit-Weiskopf
798: // mA and mB are the masses of daughters (GeV)
799: std::complex<double> CBW_BF(double m2, double M0, double Gamma, int J, double mA, double mB) {
800:   const double u     = msqrt((m2 - pow2(mA + mB)) * (m2 - pow2(mA - mB))) / (2 * msqrt(m2));
801:   const double d     = msqrt((M0 * M0 - pow2(mA + mB)) * (M0 * M0 - pow2(mA - mB))) / (2 * M0);
802:   const double Bfactor = pow(u / d, 2 * J + 1);
803:   return -1.0 / (m2 - M0 * M0 + zi * Gamma * M0 * M0 / msqrt(m2) * Bfactor);
804: }
805:
806: // 4. Spin dependent Relativistic Breit-Wigner (should not be used blindly!)
807: //
808: // [REFERENCE: Alwall el al., arxiv.org/abs/1402.1178]
809: std::complex<double> CBW_JR(double m2, double M0, double Gamma, double J) {
810:   const std::complex<double> denom = (m2 - M0 * M0 + zi * M0 * Gamma);
811:
812:   if (static_cast<int>(J) == 0) {  // J = 0
813:     return -1.0 / denom;
814:   } else if (static_cast<int>(J * 2) == 1) {  // J = 1/2
815:     return -2.0 * msqrt(m2) / denom;
816:   } else if (static_cast<int>(J) == 1) {  // J = 1
817:     return -(1.0 - m2 / (M0 * M0)) / denom;
818:   } else if (static_cast<int>(J * 2) == 3) {  // J = 3/2
819:     return -(2.0 / 3.0) * msqrt(m2) * (1.0 - m2 / (M0 * M0)) / denom;
820:   } else if (static_cast<int>(J) == 2) {  // J = 2
821:     return -(7.0 / 6.0 - (4.0 / 3.0) * (m2 / (M0 * M0)) +
822:              (2.0 / 3.0) * (m2 * m2) / (gra::math::pow4(M0))) /
823:            denom;
824:   } else {
825:     return -1.0 / denom;  // Too high spin
826:   }
827: }
828:
829: }  // namespace form
830: }  // namespace gra
```

831:


```
  1: // Gamma-Gamma Amplitudes
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: // C++
  7: #include <complex>
  8: #include <random>
  9: #include <vector>
 10:
 11: // Own
 12: #include "Graniitti/MForm.h"
 13: #include "Graniitti/MGamma.h"
 14: #include "Graniitti/MGlobals.h"
 15: #include "Graniitti/MKinematics.h"
 16: #include "Graniitti/MMath.h"
 17: #include "Graniitti/MPDG.h"
 18: #include "Graniitti/MQED.h"
 19: #include "Graniitti/MRegge.h"
 20: #include "Graniitti/MSpin.h"
 21:
 22:
 23: using gra::math::PI;
 24: using gra::math::abs2;
 25: using gra::math::msqrt;
 26: using gra::math::pow2;
 27: using gra::math::pow3;
 28: using gra::math::pow4;
 29: using gra::math::zi;
 30:
 31: using namespace gra::form;
 32:
 33: namespace gra {
 34:
 35: // Constructor
 36: MGamma::MGamma(gra::LORENTZSCALAR &lts, const std::string &modelfile) {
 37:   // ## MULTITHREADING LOCK NEEDED FOR THE INITIALIZATION ##
 38:   gra::g_mutex.lock();
 39:
 40:   // Monopolium process
 41:   if (!PARAM_MONOPOLE::initialized) {
 42:     try {
 43:       PARAM_MONOPOLE::M0 = lts.PDG.FindByPDG(PDG::PDG_monopole).mass;
 44:       PARAM_MONOPOLE::ReadParameters(modelfile);
 45:     } catch (...) {
 46:       gra::g_mutex.unlock();  // need to release here, otherwise get infinite lock
 47:       throw;
 48:     }
 49:   }
 50:   gra::g_mutex.unlock();
 51: }
 52:
 53: // ============================================================================
 54: // (yy -> fermion-antifermion pair)
 55: // yy -> e+e-, mu+mu-, tau+tau-, qqbar or Monopole-Antimonopole (spin-1/2
 56: // monopole)
 57: // production
 58: //
 59: //
 60: // This is the same amplitude as yy -> e+e- (two diagrams)
 61: // obtained by crossing e+e- -> yy annihilation, see:
 62: //
 63: // http://theory.sinp.msu.ru/comphep_old/tutorial/QED/node4.html
 64: //
 65: // beta = v/c of the monopole (or antimonopole) in their CM system
 66: //
 67: //
 68: // ************************************************************
 69: // Dirac quantization condition:
 70: //
 71: // g = 2\pi \hbar n / (\mu_0 e),    where n = 1,2,3,...
 72: // where alpha = e^2/(4*PI) is the running QED coupling
 73: //
 74: //
 75: // Numerical values:
 76: // alpha_g  = g^2 / (4*PI) ~ 34  (when n = 1)
 77: // alpha_em = e^2 / (4*PI) ~ 1/137
 78: // ************************************************************
 79: //
 80: // From lepton pair to monopole pair:
 81: // replace e -> g*beta
 82: //
 83: // 16 * pow2(PI * alpha_EM) -> pow(g*beta, 4)
```



```
 84: //
 85: // ************************************************************
 86: // Coupling schemes:
 87: //
 88: // Dirac:     alpha_g = g^2/(4pi)
 89: // Beta-dirac alpha_g = (g*beta)^2 / (4pi)
 90: //
 91: // [REFERENCE: Rajantie, physicstoday.scitation.org/doi/pdf/10.1063/PT.3.3328]
 92: // [REFERENCE: Dougall, Wick, arxiv.org/abs/0706.1042]
 93: // [REFERENCE: Reis, Sauter, arxiv.org/abs/1707.04170v1]
 94: //
 95: double MGamma::yyffbar(gra::LORENTZSCALAR &lts) {
 96:   // QED couplings
 97:   double        COUPL         = 16.0 * pow2(gra::math::PI * qed::alpha_QED());  // = e^4
 98:   const double mass           = lts.decaytree[0].p4.M();  // lepton, quark (or monopole) mass
 99:   const double mass2          = pow2(mass);
100:   const bool   MONOPOLE_MODE = (lts.decaytree[0].p.pdg == PDG::PDG_monopole) ? true : false;
101:
102:   if (PARAM_MONOPOLE::gn < 1) {
103:     throw std::invalid_argument("MGamma::yyffbar: Parameter Dirac n less than 1");
104:   }
105:
106:   // Amplitude squared
107:   double amp2 = 0.0;
108:
109:   // Monopole-Antimonopole coupling
110:   if (MONOPOLE_MODE) {
111:     static const double g = 2.0 * math::PI * PARAM_MONOPOLE::gn / qed::e_QED();
112:
113:     if (PARAM_MONOPOLE::coupling == "beta-dirac") {
114:       // Calculate beta (velocity)
115:       // const M4Vec px = pfinal[3] + pfinal[4];
116:       // M4Vec p3 = pfinal[3];
117:       // LorentzBoost(px, px.M(), p3, -1); // // Boost to the
118:       // CM frame of M\bar{M} pair
119:       // double beta = std::sqrt( p3.Px()*p3.Px() +
120:       // p3.Py()*p3.Py() + p3.Pz()*p3.Pz()) / p3.E();
121:
122:       // Faster way
123:       const double beta = msqrt(1.0 - 4.0 * pow2(PARAM_MONOPOLE::M0) / lts.s_hat);
124:       COUPL             = pow4(g) * beta);
125:     } else if (PARAM_MONOPOLE::coupling == "dirac") {
126:       COUPL = pow4(g);
127:     } else {
128:       throw std::invalid_argument("MGamma::yyffbar: Unknown PARAM_MONOPOLE::coupling " +
129:                                   PARAM_MONOPOLE::coupling);
130:     }
131:
132:     // QED tree level amplitude squared |M|^2, spin averaged and
133:     // summed
134:     amp2 = 2.0 * COUPL *
135:            ((lts.u_hat - mass2) / (lts.t_hat - mass2) + (lts.t_hat - mass2) / (lts.u_hat - mass2) +
136:             1.0 -
137:             pow2(1.0 + (2.0 * mass2) / (lts.t_hat - mass2) + (2.0 * mass2) / (lts.u_hat - mass2)));
138:
139:     /*
140:       // FeynCalc result, less simplified, but exactly same result
141:       as above
142:       amp2 = 2 * COUPL  *
143:                    ( pow4(mass)*(3*pow2(lts.t_hat) +
144:       14*lts.t_hat*lts.u_hat + 3*pow2(lts.u_hat))
145:                      -mass2*(pow3(lts.t_hat) +
146:       7*pow2(lts.t_hat)*lts.u_hat + 7*pow2(lts.u_hat)*lts.t_hat +
147:       pow3(lts.u_hat) )
148:                      -6*pow8(mass) +
149:       lts.t_hat*lts.u_hat*(pow2(lts.t_hat) + pow2(lts.u_hat)) )/ (
150:       pow2(lts.t_hat - mass2) * pow2(lts.u_hat - mass2) );
151:     */
152:
153:     // printf("%0.15f %0.15f \n", amp2, amp2_);
154:
155:   } else {  // MADGRAPH/HELAS, all helicity amplitudes individually -> for
156:     // the screening loop
157:
158:     return AmpMG5_yy_ll.CalcAmp2(lts, 0.0);
159:   }
160:
161:   // ----------------------------------------------------------------------
162:   // quark pair (charge 1/3 or 2/3), apply charge and color factors
163:   if (std::abs(lts.decaytree[0].p.pdg) <= 6) {  // we have a quark
164:
165:     const double Q           = lts.decaytree[0].p.chargeX3 / 3.0;
166:     const double NC          = 3.0;  // quarks come in three colors
```



```
167:     const double factor      = pow4(Q) * NC;
168:     const double sqrt_factor = msqrt(factor);  // sqrt to "amplitude level"
169:
170:     // amplitude squared
171:     amp2 *= factor;
172:
173:     // helicity amplitudes
174:     for (const auto &i : aux::indices(lts.hamp)) { lts.hamp[i] *= sqrt_factor; }
175:   }
176:
177:   // ----------------------------------------------------------------------
178:   // For screening loop (approximation)
179:   lts.hamp = {msqrt(amp2)};
180:   // ----------------------------------------------------------------------
181:
182:   return amp2;
183: }
184:
185: // ------------------------------------------------------------------------------------------
186: // A Monopolium (monopole-antimonopole) bound state process
187: //
188: // See e.g.
189: //
190: // [REFERENCE: Preskill,
191: // http://www.theory.caltech.edu/~preskill/pubs/preskill-1984-monopoles.pdf]
192: // [REFERENCE: Epele, Franchiotti, Garcia, Canal, Vento,
193: // https://arxiv.org/abs/hep-ph/0701133v2]
194: // [REFERENCE: Barrie, Sugamoto, Yamashita, https://arxiv.org/abs/1607.03987v3]
195: // [REFERENCE: Fanchiotti, Canal, Vento, https://arxiv.org/pdf/1703.06649.pdf]
196: // [REFERENCE: Reis, Sauter, https://arxiv.org/abs/1707.04170v1]
197: //
198: // ************************************************************
199: // Dirac quantization condition:
200: //
201: // g = 2\pi \hbar / (\mu_0 e) n,     where n = 1,2,3,...
202: // where alpha = e^2/(4*PI) is the running QED coupling
203: //
204: //
205: // Numerical values:
206: // alpha_g  = g^2 / (4*PI) ~ 34  (when n = 1)
207: // alpha_em = e^2 / (4*PI) ~ 1/137
208: // ************************************************************
209: //
210: double MGamma::yyMP(gra::LORENTZSCALAR &lts) const {
211:   // Lazy printing
212:   gra::g_mutex.lock();
213:   PARAM_MONOPOLE::PrintParameters(lts.sqrt_s);
214:   gra::g_mutex.unlock();
215:
216:   // Monopolium nominal mass and width parameters
217:   static const double M = 2.0 * PARAM_MONOPOLE::M0 + PARAM_MONOPOLE::EnergyMP(PARAM_MONOPOLE::En);
218:   static const double Gamma_M = PARAM_MONOPOLE::Gamma0;
219:
220:   if (M < 0) {
221:     throw std::invalid_argument(
222:         "MGamma::yyMP: Increase ladder level parameter En. Monopolium "
223:         "nominal mass " +
224:         std::to_string(M) + " < 0!");
225:   }
226:   if (PARAM_MONOPOLE::gn < 1) {
227:     throw std::invalid_argument("MGamma::yyMP: Parameter Dirac n less than 1");
228:   }
229:
230:   // Two coupling scenarios:
231:   static const double g    = 2.0 * math::PI * PARAM_MONOPOLE::gn / qed::e_QED();
232:   double              beta = 0.0;
233:
234:   if (PARAM_MONOPOLE::coupling == "beta-dirac") {
235:     beta = msqrt(1.0 - pow2(M) / lts.s_hat);
236:   } else if (PARAM_MONOPOLE::coupling == "dirac") {
237:     beta = 1.0;
238:   } else {
239:     throw std::invalid_argument("MGamma::yyMP: Unknown PARAM_MONOPOLE::coupling " +
240:                                 PARAM_MONOPOLE::coupling);
241:   }
242:
243:   // Magnetic coupling
244:   const double alpha_g = pow2(beta * g) / (4.0 * math::PI);
245:
246:   // Running width
247:   const double Gamma_E = PARAM_MONOPOLE::GammaMP(PARAM_MONOPOLE::En, alpha_g);
248:
249:   // printf("alpha_g = %0.3E, Gamma_E = %0.3E, Gamma_M = %0.3E, Psi_MP = %0.3E
```

```
250:   // \n",
251:   //    alpha_g, Gamma_E, Gamma_M, PARAM_MONOPOLE::PsiMP(PARAM_MONOPOLE::n));
252:
253:   // Normalization
254:   double norm = 2.0 * math::PI * M * M;
255:
256:   // Sub process
257:   double amp2 = norm * (Gamma_E * Gamma_M) / (pow2(lts.s_hat - M * M) + pow2(M * Gamma_M));
258:
259:   // ------------------------------------------------------------------
260:   // For screening loop (approximation)
261:   lts.hamp = (msqrt(amp2));
262:   // ------------------------------------------------------------------
263:
264:   return amp2;
265: }
266:
267: // ------------------------------------------------------------------------------------------
268: // Gamma-Gamma to SM-Higgs 0++ helicity amplitudes
269: //
270: // Generic narrow width yy -> X cross section in terms of partial decay widths:
271: //
272: // \sigma(yy -> X) = 8\pi^2/M_X (2J+1) \Gamma(X -> yy) \delta(shat - M_X^2) (1 + h1h2)
273: //                 = (8 * \pi)  (2J+1) \Gamma(X -> yy) \Gamma_X (1 + h1h2) /
274: //                   ((shat - M_X^2)^2 + M_X^2\Gamma_X^2),
275: //
276: // where h1,h2 = +- gamma helicities
277: //
278: // [REFERENCE: Khoze, Martin, Ryskin, https://arxiv.org/abs/hep-ph/0111078]
279: // [REFERENCE: Bernal, Lopez-Val, Sola, https://arxiv.org/pdf/0903.4978.pdf]
280: //Â [REFERENCE: Enterria, Lansberg,
281: // https://www.slac.stanford.edu/pubs/slacpubs/13750/slac-pub-13786.pdf]
282: //
283: double MGamma::yyHiggs(gra::LORENTZSCALAR &lts) const {
284:   lts.hamp.resize(4);
285:
286:   const double M        = 125.18;           // Higgs mass measured (GeV)
287:   const double Gamma    = 0.00415;          // Higgs total width calculated (GeV)
288:   const double Gamma_yy = 2.27E-3 * Gamma;  // Higgs to gamma-gamma calculated (GeV)
289:
290:   // Normalization
291:   const double norm = 2.0 * math::PI * M * M;
292:
293:   // Effective helicity amplitudes
294:   lts.hamp[0] =
295:       msqrt(norm * Gamma_yy * Gamma * (1 + 1) / (pow2(lts.s_hat - M * M) + pow2(M * Gamma)));  // --
296:   lts.hamp[1] = 0.0;                                                                           // -+
297:   lts.hamp[2] = 0.0;                                                                           // +-
298:   lts.hamp[3] = lts.hamp[0];                                                                   // ++
299:
300:   // Sum over helicity amplitudes squared
301:   double amp2 = 0.0;
302:   for (const auto &i : aux::indices(lts.hamp)) { amp2 += gra::math::abs2(lts.hamp[i]); }
303:   amp2 /= 4;  // Initial state polarization average
304:
305:   return amp2;
306: }
307:
308: // ========================================================================
309: // Simple matrix element ansatz for gamma-gamma resonances
310: //
311: // ===========
312: //       g
313: //       g
314: //       x--->
315: //       g
316: //       g
317: // ===========
318: //
319: // For Narrow-Width approximations, see:
320: //
321: // [REFERENCE: Uhlemann, Kauer, Narrow-width approximation accuracy,
322: // https://arxiv.org/pdf/0807.4112.pdf]
323: //
324: double MGamma::yyX(gra::LORENTZSCALAR &lts, gra::PARAM_RES &resonance) const {
325:   // Factor of 2 x from (identical) initial state boson statistics
326:   std::complex<double> A_prod =
327:       2.0 * gra::form::CBW(lts, resonance) *
328:       PARAM_REGGE::ResonanceFormFactor(lts.m2, pow2(resonance.p.mass), resonance.g_FF);
329:
330:   // Production and Decay amplitude
331:   const std::complex<double> A_spin =
332:       spin::ProdAmp(lts, resonance) * spin::DecayAmp(lts, resonance);
```



```
333:
334:   // Total
335:   const std::complex<double> A = A_prod * A_spin;
336:
337:   // ------------------------------------------------------------------
338:   // For screening loop
339:   lts.hamp = (A);
340:   // ------------------------------------------------------------------
341:
342:   return abs2(A);
343: }
344:
345: namespace PARAM_MONOPOLE {
346:
347: int         En       = 0;       // Bound state energy level
348: double      M0       = 0.0;     // Monopole mass
349: double      Gamma0   = 0.0;     // Monopolium width
350: std::string coupling = "null";  // Coupling scenarios
351: int         gn       = 0;       // Dirac charge 1,2,3,...
352:
353: bool printed     = false;
354: bool initialized = false;
355:
356: // Functions as a solution to radial Schroedinger equation with Coulomb type
357: // potential
358: // V(r) ~= -g^2/(4pi) * 1/r
359: //
360: // Monopolium binding energy eigenvalues
361: // -> Monopolium system sits at lower energy than monopole + antimonopole
362: //
363: // Monopolium running width
364: double GammaMP(double n, double alpha_g) {
365:   return (8.0 * PI * pow2(alpha_g)) / pow2(PARAM_MONOPOLE::M0) * math::abs2(PsiMP(n));
366: }
367:
368: // Binding energy
369: double EnergyMP(double n) {
370:   return -pow2(1 / (8.0 * qed::alpha_QED())) * PARAM_MONOPOLE::M0 / (n * n);
371:   // return -2.0*M0/15.0; // DEBUG
372: }
373:
374: // Monopolium wavefunction in the origin of the bound system
375: double PsiMP(double n) {
376:   return (1.0 / msqrt(PI)) * std::pow(PARAM_MONOPOLE::M0 / (8.0 * qed::alpha_QED() * n), 3.0 / 2.0);
377: }
378:
379: void PrintParameters(double sqrts) {
380:   if (!printed) {
381:     aux::PrintBar("*");
382:     std::cout << rang::style::bold << "Monopolium process parameters:" << rang::style::reset
383:               << std::endl
384:               << std::endl;
385:
386:     printf("~ M0      = %0.3f GeV \n", PARAM_MONOPOLE::M0);
387:     printf("~ Gamma0 = %0.3f GeV \n", PARAM_MONOPOLE::Gamma0);
388:     printf("~ En      = %d \n", PARAM_MONOPOLE::En);
389:
390:     const double M = 2 * PARAM_MONOPOLE::M0 + EnergyMP(PARAM_MONOPOLE::En);
391:     printf(
392:         "\nGives monopolium mass: M = %0.3f GeV (Binding Energy = %0.3f "
393:         "GeV) \n\n",
394:         M, M - 2 * PARAM_MONOPOLE::M0);
395:
396:     // Check we have enough energy
397:     if (M > sqrts) {
398:       std::string str =
399:           "PARAM_MONOPOLE::PrintParam: FATAL error "
400:           "Monopolium mass > CMS energy!";
401:       throw std::invalid_argument(str);
402:     }
403:
404:     std::cout << "Dirac charge = " << PARAM_MONOPOLE::gn << std::endl;
405:     std::cout << "Coupling scheme = " << PARAM_MONOPOLE::coupling << std::endl;
406:     aux::PrintBar("*");
407:     std::cout << std::endl;
408:
409:     printed = true;
410:   }
411: }
412:
413:
414: // Read parameters from file
415: void ReadParameters(const std::string &modelfile) {
```



```
416:   try {
417:     const std::string data = gra::aux::GetInputData(modelfile);
418:     nlohmann::json    j    = nlohmann::json::parse(data);
419:
420:     // JSON block identifier
421:     const std::string XID = "PARAM_MONOPOLE";
422:
423:     // Monopole production
424:     PARAM_MONOPOLE::coupling = j.at(XID).at("coupling");
425:     PARAM_MONOPOLE::gn       = j.at(XID).at("gn");
426:     PARAM_MONOPOLE::En       = j.at(XID).at("En");
427:     PARAM_MONOPOLE::Gamma0   = j.at(XID).at("Gamma0");
428:
429:     initialized = true;
430:   } catch (...) {
431:     std::string str = "PARAM_MONOPOLE::ReadParameters: Error parsing " + modelfile +
432:                       " (Check for extra/missing commas)";
433:     throw std::invalid_argument(str);
434:   }
435: }
436:
437: }  // namespace PARAM_MONOPOLE
438:
439:
440: }  // namespace gra
441:
```



```
  1: // Regge Amplitudes
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: // C++
  7: #include <complex>
  8: #include <random>
  9: #include <vector>
 10:
 11: // Own
 12: #include "Graniitti/MAux.h"
 13: #include "Graniitti/MEikonal.h"
 14: #include "Graniitti/MForm.h"
 15: #include "Graniitti/MGlobals.h"
 16: #include "Graniitti/MKinematics.h"
 17: #include "Graniitti/MMath.h"
 18: #include "Graniitti/MRegge.h"
 19: #include "Graniitti/MSpin.h"
 20:
 21: // Libraries
 22: #include "json.hpp"
 23: #include "rang.hpp"
 24:
 25: using gra::math::msqrt;
 26: using gra::math::pow2;
 27: using gra::math::pow3;
 28: using gra::math::zi;
 29:
 30: using gra::aux::indices;
 31: using namespace gra::form;
 32:
 33:
 34: namespace gra {
 35:
 36: // Constructor
 37: MRegge::MRegge(gra::LORENTZSCALAR &lts, const std::string &modelfile) {
 38:   // Initialization
 39:   gra::g_mutex.lock();
 40:   if (!PARAM_REGGE::initialized && lts.decaytree.size() != 0) {
 41:     try {
 42:       const int PDG = std::abs(lts.decaytree[0].p.pdg);
 43:       PARAM_REGGE::ReadParameters(PDG, modelfile);
 44:
 45:       // Construct 4 and 6 body Regge Ladder leg combinatorial permutations
 46:       const int mode = 1;  // charged permutations +- (FIXED FOR NOW)
 47:       permutations4  = gra::math::GetAmpPerm(4, mode);
 48:       permutations6  = gra::math::GetAmpPerm(6, mode);
 49:
 50:     } catch (...) {
 51:       gra::g_mutex.unlock();  // need to release here, otherwise get infinite lock
 52:       throw;
 53:     }
 54:   }
 55:   gra::g_mutex.unlock();
 56: }
 57:
 58: // Triple-Regge (Pomeron) limits
 59: //
 60: // discontinuity line (cut line)
 61: //         |
 62: //         |
 63: //
 64: //         .
 65: //         .
 66: // SD:
 67: //
 68: // a =gN= a a =gN= a
 69: //     *     *
 70: //  i   *     *   j
 71: //        *3P*
 72: //       k *  M_X^2
 73: //         *
 74: // a =====gN===== a
 75: //
 76: //
 77: // DD:
 78: //
 79: // a =====gN===== a
 80: //    k1  *
 81: //         *  M_Y^2
 82: //        *3P*
 83: //   i  *     *   j
```

```
//       *    *
//        *3P*
//         *  M_X^2
//    k2  *
// a =====gN====== a
//
//
// CD:
//
// a =gN= a  a =gN= a
//      *     *
//  i1  *     *  j1
//        *3P*
//      k  *  M_X^2
//         *
//        *3P*
//  i2  *     *  j2
//      *     *
// a =gN= a  a =gN= a
//
//
//
// [REFERENCE: Gribov, A Reggeon Diagram Technique, Soviet JETP, 1968,
// jetp.ac.ru/cgi-bin/dn/a_026_02_0414.pdf]
//
// Strong coupling in the Pomeranchuk pole problem
// [REFERENCE: Gribov, Migdal, Sov. Phys. JETP 28(4), 784-795 (1968)]
// [REFERENCE: Muller, 1972]
//
// For different forms of triple Pomeron coupling (weak/strong, scalar/vector):
// [REFERENCE: Luna, Khoze, Martin, Ryskin, arxiv.org/abs/1005.4864v1]
//
std::complex<double> MRegge::ME2(gra::LORENTZSCALAR &lts, int mode) const {
  const double s0 = 1.0;  // Energy scale GeV^2

  // Pomeron trajectory intercept + 1
  const double alpha_0 = 1.0 + PARAM_SOFT::DELTA_P;

  // Triple legs
  const double alpha_t_i = S3PomAlpha(lts.t);
  const double alpha_t_j = alpha_t_i;

  std::complex<double> A(0, 0);

  if (mode == 1) {  // Elastic

    // Single Pomeron exchange
    // (this calls eikonal pomeron without eikonalization, just for test, use loop on)
    return MEikonal::SingleAmpElastic(lts.s, lts.t, 1);

  } else if (mode == 2) {  // SD triple Pomeron

    // Which proton is excited
    const double MX_2 = (lts.ss[1][1] > 1.0) ? lts.ss[1][1] : lts.ss[2][2];

    const double amp2 = pow3(PARAM_SOFT::gN_P)  // Proton-Pomeron coupling ^ 3
                        * pow2(S3F(lts.t))      // Proton form factor ^ 2
                        * PARAM_SOFT::g3P        // Triple-Pomeron coupling

                        * std::pow(lts.s / MX_2,
                                   alpha_t_i + alpha_t_j)  // LEFT + RIGHT LEG

                        * std::pow(MX_2 / s0, alpha_0);  // DISCONTINUITY PART

    // recover amplitude, we had the expression at cross section level
    A = gra::form::ReggeEta(alpha_t_i, 1) * msqrt(amp2);

  } else if (mode == 3) {  // DD triple Pomeron, note NO proton form
    // factors (both proton dissociated)

    const double MX_2 = lts.ss[1][1];
    const double MY_2 = lts.ss[2][2];

    const double amp2 = pow2(PARAM_SOFT::gN_P)    // Proton-Pomeron ^ 2
                        * pow2(PARAM_SOFT::g3P)  // Triple-Pomeron coupling ^ 2

                        * std::pow((lts.s * s0) / (MX_2 * MY_2),
                                   alpha_t_i + alpha_t_j)  // LEFT + RIGHT LEG

                        * std::pow(MX_2 / s0, alpha_0)   // DISCONTINUITY PART
                        * std::pow(MY_2 / s0, alpha_0);  // DISCONTINUITY PART

    // recover amplitude, we had the expression at cross section level
```



```
    A = gra::form::ReggeEta(alpha_t_i, 1) * msqrt(amp2);
  } else {
    throw std::invalid_argument("MRegge::ME2: Unknown mode parameter: " + std::to_string(mode));
  }

  // -----------------------------------------------------------------
  // For screening loop
  lts.hamp = {A};
  // -----------------------------------------------------------------

  return A;
}

// ======================================================================
// Helicity matrix element for Pomeron-Pomeron resonances
//
// [THIS IS UNDER CONSTRUCTION!]
//
//  lambda1    lambda3
//  ==========>
//             *
//             *
//            x----> lambda_h (5)
//             *
//             *
//  ==========>
//  lambda2    lambda4
//
// [Check Witten-Sakai-Sugimoto model based holographic eta,eta':
// [arxiv.org/abs/1406.7010]
//
std::complex<double> MRegge::ME3HEL(gra::LORENTZSCALAR &lts, gra::PARAM_RES &resonance) const {
  const int J = resonance.p.spinX2 / 2.0;

  // -----------------------------------------------------------------
  unsigned int      number = 0;
  const unsigned int N      = 16 * (2 * J + 1);

  // C++11 guarantees a local static variable initialization thread safety!
  static std::vector<std::vector<double>> lambda(N, std::vector<double>(5));

  for (double i = 0; i < 2; ++i) {                    // Proton 1
    for (double j = 0; j < 2; ++j) {                  // Proton 2
      for (double k = 0; k < 2; ++k) {                // Proton 3
        for (double l = 0; l < 2; ++l) {              // Proton 4
          for (double m = 0; m < (2 * J + 1); ++m) {  // Central Boson

            lambda[number] = {i - 0.5, j - 0.5, k - 0.5, l - 0.5, m - static_cast<double>(J)};
            ++number;
          }
        }
      }
    }
  }

  // -----------------------------------------------------------------
  // Proton / Dissociative system vertices
  double FF_A = 0.0;
  double FF_B = 0.0;
  PomPomProtonVertex(lts, FF_A, FF_B);

  // Forward proton deltaphi
  // const double dphi = lts.pfinal[1].DeltaPhi(lts.pfinal[2]);
  // const double dphi = lts.pfinal[1].Phi();

  // Boost propagator to the system rest frame
  M4Vec q1 = lts.q1;
  gra::kinematics::LorentzBoost(lts.pfinal[0], lts.pfinal[0].M(), q1, -1);
  const double dphi = q1.Phi();

  // Now loop over all helicity amplitudes
  // Common factor all amplitudes
  std::complex<double> common =
      FF_A * PropOnly(lts.s1, lts.t1) * CBW(lts, resonance) * resonance.g *
      PARAM_REGGE::ResonanceFormFactor(lts.m2, pow2(resonance.p.mass), resonance.g_FF) *
      PropOnly(lts.s2, lts.t2) * FF_B;

  lts.hamp.clear();
  for (std::size_t i = 0; i < N; ++i) {
    // Apply upper vertex helicity conservation
    if (g_Vertex(lts.t1, lambda[i][0], lambda[i][2]) !=
        std::pow(-1, lambda[i][0] - lambda[i][2]) * g_Vertex(lts.t1, lambda[i][0], lambda[i][2])) {
```



```
      continue;
    }
    // Apply lower vertex helicity conservation
    if (g_Vertex(lts.t2, lambda[i][1], lambda[i][3]) !=
        std::pow(-1, lambda[i][1] - lambda[i][3]) *
            g_Vertex(lts.t2, -lambda[i][1], -lambda[i][3])) {
      continue;
    }

    // Spin density matrix weight for this helicity
    const int                  index = lambda[i][4] + resonance.p.spinX2;  // Index the diagonal
    const std::complex<double> rhoweight =
        resonance.p.spinX2 != 0 ? resonance.rho[index][index] : 1.0;

    // Calculate amplitude
    const std::complex<double> amp =
        rhoweight * g_Vertex(lts.t1, lambda[i][0], lambda[i][2]) *
        gik_Vertex(lts.t1, lts.t2, dphi, lambda[i][4], resonance.p.spinX2 / 2.0, resonance.p.P,
                   resonance.JMAX) *
        common;
    g_Vertex(lts.t2, lambda[i][1], lambda[i][3]);

    // std::cout << amp << " :: " << gra::math::abs2(amp) << std::endl;
    lts.hamp.push_back(amp);
  }

  // 1/4 (|H1|^2 + |H2|^2 + ... + |HN|^2)
  double amp2 = 0.0;
  for (std::size_t i = 0; i < lts.hamp.size(); ++i) { amp2 += gra::math::abs2(lts.hamp[i]); }
  amp2 /= 4;  // Initial state average

  return msqrt(amp2);  // we expect amplitude
}

// This function is used for parity conservation check:
//
// [ gamma_{m1,m2}^\lambda_h = (-1)^\lambda_h \xi_3 \gamma_{-m1-m2}^-\lambda ]
//
// Pomeron i  (Parity P, Signature sigma)
// Pomeron k  (Parity P, Signature sigma)
// Resonance  (Spin J, Parity P)
//
// "Naturality" \def= Reggeon Parity x Reggeon Regge Signature
//
int MRegge::xi3(int J, int P, int P_i, int sigma_i, int P_k, int sigma_k) const {
  return P * std::pow(-1, J) * P_i * sigma_i * P_k * sigma_k;
}

// ------------------------------------------------------------------------------
// Proton-Pomeron-Proton vertex with helicity transition lambda_i-> lambda_f
//
// initial > final
//    -1/2 > -1/2
//    -1/2 >  1/2
//     1/2 > -1/2
//     1/2 >  1/2
//
// ~ works for small |t|
//
// g_{\lambda_1,\lambda_3}(t) = (-1)^{\lambda_1 - \lambda_3} \xi_1
// g_{-\lambda_1-\lambda_3}(t)
// \xi_1 = \eta_1\eta_3 (-1)^{S_1 - S_3} P_i \sigma_i
//
double MRegge::g_Vertex(double t, double lambda_i, double lambda_f) const {
  return std::pow(-t, std::abs(lambda_i - lambda_f) / 2);
}

// Central vertex subfunction: For small |t|, |t2|
//
// Selection rules:
// 1. m1 - m2 = lambda_h (angular momentum)
// 2. parity conservation
//
double MRegge::gammaLambda(double t1, double t2, double m1, double m2) const {
  // Parity sign flip under (m1,m2) <-> (-m1,-m2)
  const int P = std::pow(-1, math::sign(m1 - std::abs(m1)) * math::sign(m2 - std::abs(m2)));
  return P * std::pow(-t1, std::abs(m1) / 2) * std::pow(-t2, std::abs(m2) / 2);
}

// Central vertex g_ik(t1,t2,\phi)^{\lambda_h} (central resonance J, P)
//
// Selection rules:
// 1. lambda_h = m1 - m2 (angular momentum)
```



```
// 2. parity conservation
//
// Naturality: intrinsic parity x (-1)^J       (bosons)
//             intrinsic parity x (-1)^(J-1/2) (fermions)
//
//
std::complex<double> MRegge::gik_Vertex(double t1, double t2, double dphi, int lambda_h, int J,
                                        int P, int JMAX) const {
  // (we handle only Pomerons here for now, no secondary Reggeons - trivial
  // extension)
  const int pom1_P     = 1;  // Pomeron 1 parity +
  const int pom1_sigma = 1;  // Pomeron 1 signature +
  const int pom2_P     = 1;  // Pomeron 2 parity +
  const int pom2_sigma = 1;  // Pomeron 2 signature +

  const int           xi3val = xi3(J, P, pom1_P, pom1_sigma, pom2_P, pom2_sigma);
  std::complex<double> sum    = 0.0;

  for (int m1 = -JMAX; m1 <= JMAX; ++m1) {  // -\inf < m_1 < inf

    // Helicity relation lambda_h = m1 - m2
    const int m2 = m1 - lambda_h;

    // Parity conservation
    const double gamma_L = gammaLambda(t1, t2, m1, m2);
    const double gamma_R = std::pow(-1, lambda_h) * xi3val * gammaLambda(t1, t2, -m1, -m2);

    if (math::sign(gamma_L) != math::sign(gamma_R)) continue;

    // Analytical continuation
    sum += std::exp(gra::math::zi * static_cast<double>(m1) * dphi) * gamma_L;
  }
  // std::cout << std::endl;
  return sum;
}

// ======================================================================
// Regge matrix element ansatz for 2->4 continuum spectrum
// with sign, one can choose the relative sign between sub-t and sub-u
// amplitudes -> e.g. baryons (fermions) (could) have different effective sign
// than mesons (bosons).
//
//  ======F=======>
//         *
//         * P
//         *
//         ff ----> pi+/K+,p,rho0 ...
//         |
//         ff ----> pi-/K-,pbar,rho0 ...
//         *
//         * P
//         *
//  ======F=======>
//
// For similar Regge constructions, see:
//
// [REFERENCE: Azimov, Khoze, Levin, Ryskin, Sov.J.Nucl.Phys.21, 215 (1975)]
// [REFERENCE: Pumplin, Henyey, Nucl.Phys. B117, 377 (1976)]
// [REFERENCE: Lebiedowicz, Szczurek, arxiv.org/abs/0912.0190]
// [REFERENCE: Harland-Lang, Khoze, Ryskin, arxiv.org/abs/1312.4553]
//
std::complex<double> MRegge::ME4(gra::LORENTZSCALAR &lts, double sign) const {
  // Offshell propagator nominal mass^2 read from the outgoing particle PDG mass
  const double M2_ = pow2(lts.decaytree[0].p.mass);

  // Create function pointers
  double (*ff)(double, double);
  double (*prop)(double, double);

  if (sign > 0) {  // positive sign amplitudes

    ff   = &PARAM_REGGE::Meson_FF;
    prop = &PARAM_REGGE::Meson_prop;
  } else {  // negative sign (alternative model spin-statistics) amplitude

    ff   = &PARAM_REGGE::Baryon_FF;
    prop = &PARAM_REGGE::Baryon_prop;
  }

  // Proton / Dissociative system vertices
  double FF_A = 0.0;
  double FF_B = 0.0;
  PomPomProtonVertex(lts, FF_A, FF_B);
```

```
416:
417:   // Particle-Particle-Pomeron coupling
418:   const double gpp_P = PARAM_REGGE::c[0] / PARAM_SOFT::gN_P;
419:
420:   const std::complex<double> A_t = PropOnly(lts.ss[1][3], lts.t1) * FF_A * (*ff)(lts.t_hat, M2_) *
421:                                    gpp_P * prop(lts.t_hat, M2_) * (*ff)(lts.t_hat, M2_) * gpp_P *
422:                                    PropOnly(lts.ss[2][4], lts.t2) * FF_B;
423:
424:   // sign applied here
425:   const std::complex<double> A_u =
426:       sign * PropOnly(lts.ss[1][4], lts.t1) * FF_A * (*ff)(lts.u_hat, M2_) * gpp_P *
427:       prop(lts.u_hat, M2_) * (*ff)(lts.u_hat, M2_) * gpp_P * PropOnly(lts.ss[2][3], lts.t2) * FF_B;
428:
429:   // Total amplitude
430:   const std::complex<double> A = A_t + A_u;
431:
432:   // ------------------------------------------------------------------
433:   // For screening loop
434:   lts.hamp = {A};
435:   // ------------------------------------------------------------------
436:
437:   return A;
438: }
439:
440: // ======================================================================
441: // Regge matrix element ansatz for 2->6 continuum spectrum
442: //
443: //
444: //  ======F======>
445: //        *
446: //        *
447: //       ff-----> a
448: //       |
449: //       ff-----> b
450: //        *
451: //        *
452: //       ff-----> c
453: //       |
454: //       ff-----> d
455: //        *
456: //        *
457: //  ======F======>
458: //
459: // For many similar amplitudes, see the literature of the era
460: // "pre-superstring/generalized Veneziano amplitude". For example:
461: //
462: // [REFERENCE: Bardakci, Ruegg, journals.aps.org/pr/pdf/10.1103/PhysRev.181.1884]
463: //
464: //
465: // Numerically, one may compare with:
466: //
467: // [REFERENCE: Kycia, Lebiedowicz, Szczurek, Turnau, arxiv.org/abs/1702.07572]
468: //
469: //
470: std::complex<double> MRegge::ME6(gra::LORENTZSCALAR &lts) const {
471:   // Amplitude_
472:   std::complex<double> A(0, 0);
473:
474:   // Offshell propagator masses^2 [for mixed states, such as pi+ pi- K+ K-]
475:   const double M2_A = pow2(lts.decaytree[0].p.mass);
476:   const double M2_B = pow2(lts.decaytree[2].p.mass);
477:
478:   // Create function pointers
479:   double (*ff)(double, double);
480:   double (*prop)(double, double);
481:
482:   const double sign = 1;
483:
484:   if (sign > 0) {  // positive sign amplitudes
485:
486:     ff   = &PARAM_REGGE::Meson_FF;
487:     prop = &PARAM_REGGE::Meson_prop;
488:   } else {  // negative sign (alternative model spin-statistics) amplitude
489:
490:     ff   = &PARAM_REGGE::Baryon_FF;
491:     prop = &PARAM_REGGE::Baryon_prop;
492:   }
493:
494:   // Proton / Dissociative system vertices
495:   double FF_A = 0.0;
496:   double FF_B = 0.0;
497:   PomPomProtonVertex(lts, FF_A, FF_B);
498:
```



```
499:   // Particle-Particle-Pomeron coupling
500:   const double gpp_P = PARAM_REGGE::c[0] / PARAM_SOFT::gN_P;
501:
502:   // Loop over different final state permutations (max #16)
503:   for (const auto &i : indices(permutations4)) {
504:     const unsigned int a = permutations4[i][0];
505:     const unsigned int b = permutations4[i][1];
506:     const unsigned int c = permutations4[i][2];
507:     const unsigned int d = permutations4[i][3];
508:
509:     // Calculate t-type Lorentz scalars here [no need, done already]
510:     // const double tt_ab = (pbeam1_pfinal1 - pfinal[a]).M2();
511:     // const double tt_bc = (pbeam1_pfinal1 - pfinal[a] - pfinal[b]).M2(); // by
512:     // symmetry, same as (pbeam2 - pfinal[2] - pfinal[d] - pfinal[c]).M2()
513:     // const double tt_cd = (pbeam2_pfinal2 - pfinal[d]).M2();
514:
515:     // Get Lorentz scalars
516:     const double tt_ab = lts.tt_1[a];
517:     const double tt_bc = lts.tt_xy[a][b];
518:     const double tt_cd = lts.tt_2[d];
519:
520:     const std::complex<double> subamp =
521:         PropOnly(lts.ss[1][a], lts.t1) * FF_A * (*ff)(tt_ab, M2_A) * gpp_P * prop(tt_ab, M2_A) *
522:         (*ff)(tt_bc, M2_A) * gpp_P * PropOnly(lts.ss[b][c], tt_bc) * (*ff)(tt_bc, M2_B) * gpp_P *
523:         prop(tt_cd, M2_B) * (*ff)(tt_cd, M2_B) * gpp_P * PropOnly(lts.ss[2][d], lts.t2) * FF_B;
524:
525:     A += subamp;
526:   }
527:
528:   // ------------------------------------------------------------------
529:   // For screening loop
530:   lts.hamp = {A};
531:   // ------------------------------------------------------------------
532:
533:   return A;
534: }
535:
536: // ======================================================================
537: // Regge matrix element ansatz for 2->8 continuum spectrum (generalization of
538: // 2->6)
539: //
540: //  ======F======>
541: //        *
542: //        *
543: //       ff-----> a
544: //       |
545: //       ff-----> b
546: //        *
547: //        *
548: //       ff-----> c
549: //       |
550: //       ff-----> d
551: //        *
552: //        *
553: //       ff-----> e
554: //       |
555: //       ff-----> f
556: //        *
557: //        *
558: //  ======F======>
559: //
560: std::complex<double> MRegge::ME8(gra::LORENTZSCALAR &lts) const {
561:   // Amplitude
562:   std::complex<double> A(0, 0);
563:
564:   // Offshell propagator masses^2 [for mixed states, such as pi+ pi- K+ K- pi+
565:   // pi-]
566:   const double M2_A = pow2(lts.decaytree[0].p.mass);
567:   const double M2_B = pow2(lts.decaytree[2].p.mass);
568:   const double M2_C = pow2(lts.decaytree[4].p.mass);
569:
570:   // Create function pointers
571:   double (*ff)(double, double);
572:   double (*prop)(double, double);
573:
574:   const double sign = 1;
575:
576:   if (sign > 0) {  // positive sign amplitudes
577:
578:     ff   = &PARAM_REGGE::Meson_FF;
579:     prop = &PARAM_REGGE::Meson_prop;
580:   } else {  // negative sign (alternative model spin-statistics) amplitude
581:
```



```
582:     ff   = &PARAM_REGGE::Baryon_FF;
583:     prop = &PARAM_REGGE::Baryon_prop;
584:   }
585:
586:   // Proton / Dissociative system vertices
587:   double FF_A = 0.0;
588:   double FF_B = 0.0;
589:   PomPomProtonVertex(lts, FF_A, FF_B);
590:
591:   // Particle-Particle-Pomeron coupling
592:   const double gpp_P = PARAM_REGGE::c[0] / PARAM_SOFT::gN_P;
593:
594:   // Loop over different permutations (max #288)
595:   for (const auto &i : indices(permutations6)) {
596:     const unsigned int a = permutations6[i][0];
597:     const unsigned int b = permutations6[i][1];
598:     const unsigned int c = permutations6[i][2];
599:     const unsigned int d = permutations6[i][3];
600:     const unsigned int e = permutations6[i][4];
601:     const unsigned int f = permutations6[i][5];
602:
603:     // t-type Lorentz scalars [no need here, already calculated]
604:     // const double tt_ab = (pbeam1_pfinal1 - pfinal[a]).M2();
605:     // const double tt_bc = (pbeam1_pfinal1 - pfinal[a] - pfinal[b]).M2();
606:     // const double tt_cd = (pbeam1_pfinal1 - pfinal[b] - pfinal[c]).M2();
607:     // const double tt_de = (pbeam2_pfinal1 - pfinal[c] - pfinal[d]).M2();
608:     // const double tt_ef = (pbeam2_pfinal2 - pfinal[f]).M2();
609:
610:     // Collect scalars
611:     const double tt_ab = lts.tt_1[a];
612:     const double tt_bc = lts.tt_xy[a][b];
613:     const double tt_cd = lts.tt_xy[b][c];
614:     const double tt_de = lts.tt_xy[c][d];
615:     const double tt_ef = lts.tt_2[f];
616:
617:     const std::complex<double> subamp =
618:         PropOnly(lts.ss[1][a], lts.t1) * FF_A * (*ff)(tt_ab, M2_A) * gpp_P * prop(tt_ab, M2_A) *
619:         (*ff)(tt_bc, M2_A) * gpp_P * PropOnly(lts.ss[b][c], tt_bc) * (*ff)(tt_bc, M2_B) * gpp_P *
620:         prop(tt_cd, M2_B) * (*ff)(tt_de, M2_B) * gpp_P * PropOnly(lts.ss[d][e], tt_de) *
621:         (*ff)(tt_de, M2_C) * gpp_P * prop(tt_ef, M2_C) * (*ff)(tt_ef, M2_C) * gpp_P *
622:         PropOnly(lts.ss[2][f], lts.t2) * FF_B;
623:
624:     A += subamp;
625:   }
626:
627:   // ------------------------------------------------------------------
628:   // For screening loop
629:   lts.hamp = {A};
630:   // ------------------------------------------------------------------
631:
632:   return A;
633: }
634:
635: // Proton-Proton-Pomeron elastic / inelastic vertex
636: //
637: void MRegge::PomPomProtonVertex(const gra::LORENTZSCALAR &lts, double &FF_A, double &FF_B) const {
638:   FF_A = lts.excite1 ? msqrt(PARAM_SOFT::g3P * PARAM_SOFT::gN_P) *
639:                        gra::form::S3FINEL(lts.t1, lts.pfinal[1].M2())
640:                      : PARAM_SOFT::gN_P * gra::form::S3F(lts.t1);
641:   FF_B = lts.excite2 ? msqrt(PARAM_SOFT::g3P * PARAM_SOFT::gN_P) *
642:                        gra::form::S3FINEL(lts.t2, lts.pfinal[2].M2())
643:                      : PARAM_SOFT::gN_P * gra::form::S3F(lts.t2);
644: }
645:
646: // ======================================================================
647: // Generic resonance (sub)-cross section:
648: //
649: //        \hat{\sigma} ~ \pi / \hat{s} [W_i(\hat{s}) W_f(\hat{s})] /
650: //                                  [(\hat{s} - M_0)^2 + W_f(\hat{s})^2 ]
651: //
652: // where W_{i,f} denotes effective initial (final) state factors
653: // incorporating spin and color symmetry factors, couplings and the rest of the
654: // dynamics
655: //
656: //
657: // In general, the behavior can be different in domains:
658: //
659: // 1. \hat{s} << M0^2, where M0 is the on-shell mass
660: // 2. \hat{s} ~ M0^2
661: // 3. \hat{s} >> M0^2
662: //
663: // and isolated resonances might break unitarity when \hat{s} -> inf,
664: // but resonance + continuum amplitude does not.
```



```
665:
666: // ======================================================================
667: // Simple matrix element ansatz for Pomeron-Pomeron resonances
668: //
669: // ==========
670: //       *
671: //       *
672: //       x---->
673: //       *
674: //       *
675: // ==========
676: //
677: std::complex<double> MRegge::ME3(gra::LORENTZSCALAR &lts, gra::PARAM_RES &resonance) const {
678:   // Proton / Dissociative system vertices
679:   double FF_A = 0.0;
680:   double FF_B = 0.0;
681:   PomPomProtonVertex(lts, FF_A, FF_B);
682:
683:   // s-channel
684:   // Factor 2 x from initial state (identical boson) statistics
685:   const std::complex<double> A_prod =
686:       2.0 * PropOnly(lts.s1, lts.t1) * FF_A * CBW(lts, resonance) *
687:       PARAM_REGGE::ResonanceFormFactor(lts.m2, pow2(resonance.p.mass), resonance.g_FF) *
688:       PropOnly(lts.s2, lts.t2) * FF_B;
689:
690:   // Production and Decay amplitude
691:   const std::complex<double> A_spin =
692:       spin::ProdAmp(lts, resonance) * spin::DecayAmp(lts, resonance);
693:
694:   // Flux
695:   const double V = std::pow(1.0 / lts.m2, PARAM_REGGE::omega);
696:
697:   // Full amplitude
698:   const std::complex<double> A = A_prod * A_spin * V;
699:
700:   // ------------------------------------------------------------------
701:   // For screening loop
702:   lts.hamp = {A};
703:   // ------------------------------------------------------------------
704:
705:   return A;
706: }
707:
708: // ======================================================================
709: // Simple matrix element ansatz for Odderon-Pomeron resonances
710: //
711: // ==========
712: //       O
713: //       O
714: //       x---->
715: //       P
716: //       P
717: // ==========
718: //
719: std::complex<double> MRegge::ME3ODD(gra::LORENTZSCALAR &lts, gra::PARAM_RES &resonance) const {
720:   // Proton / Dissociative system vertices
721:   double FF_A = lts.excite1 ? msqrt(PARAM_SOFT::g3P * PARAM_SOFT::gN_P) *
722:                               gra::form::S3FINEL(lts.t1, lts.pfinal[1].M2())
723:                             : PARAM_SOFT::gN_P * gra::form::S3F(lts.t1);
724:   double FF_B = lts.excite2 ? msqrt(PARAM_SOFT::g3P * PARAM_SOFT::gN_O) *
725:                               gra::form::S3FINEL(lts.t2, lts.pfinal[2].M2())
726:                             : PARAM_SOFT::gN_O * gra::form::S3F(lts.t2);
727:
728:   const std::complex<double> A1 =
729:       PropOnly(lts.s1, lts.t1) * FF_A * CBW(lts, resonance) *
730:       PARAM_REGGE::ResonanceFormFactor(lts.m2, pow2(resonance.p.mass), resonance.g_FF) *
731:       OdderonProp(lts.s2, lts.t2) * FF_B;
732:
733:   // ------------------------------------------------------------------
734:
735:   // Proton / Dissociative system vertices
736:   FF_A = lts.excite1 ? msqrt(PARAM_SOFT::g3P * PARAM_SOFT::gN_O) *
737:                        gra::form::S3FINEL(lts.t1, lts.pfinal[1].M2())
738:                      : PARAM_SOFT::gN_O * gra::form::S3F(lts.t1);
739:   FF_B = lts.excite2 ? msqrt(PARAM_SOFT::g3P * PARAM_SOFT::gN_P) *
740:                        gra::form::S3FINEL(lts.t2, lts.pfinal[2].M2())
741:                      : PARAM_SOFT::gN_P * gra::form::S3F(lts.t2);
742:
743:   const std::complex<double> A2 =
744:       OdderonProp(lts.s1, lts.t1) * FF_A * CBW(lts, resonance) *
745:       PARAM_REGGE::ResonanceFormFactor(lts.m2, pow2(resonance.p.mass), resonance.g_FF) *
746:       PropOnly(lts.s2, lts.t2) * FF_B;
747:
```

```
748:   // ------------------------------------------------------------------------
749:
750:   // Production and Decay amplitude
751:   const std::complex<double> A_spin =
752:       spin::ProdAmp(lts, resonance) * spin::DecayAmp(lts, resonance);
753:
754:   // Flux
755:   const double V = std::pow(1.0 / lts.m2, PARAM_REGGE::omega);
756:
757:   // Should sum here with negative sign if proton-antiproton initial state (anti-symmetric)
758:   const std::complex<double> A_prod = (lts.beam1.pdg == lts.beam2.pdg) ? (A1 + A2) : (A1 - A2);
759:   const std::complex<double> A      = A_prod * A_spin * V;
760:
761:   // ------------------------------------------------------------------------
762:   // For screening loop
763:   lts.hamp = {A};
764:   // ------------------------------------------------------------------------
765:
766:   return A;
767: }
768:
769:
770: // ============================================================================
771: // Simple matrix element ansatz for Photoproduction of resonances
772: //
773: // ===========
774: //       g
775: //       g
776: //      x---->
777: //       *
778: //       *
779: // ===========
780: //
781: std::complex<double> MRegge::PhotoME3(gra::LORENTZSCALAR &lts, gra::PARAM_RES &resonance) const {
782:   // Check spin
783:   if (resonance.p.spinX2 != 2) {
784:     throw std::invalid_argument("MRegge::PhotoME3: Resonance spin J = " +
785:                                 std::to_string(resonance.p.spinX2) + " (should be J = 1)!");
786:   }
787:
788:   // Resonance part
789:   const std::complex<double> common =
790:       CBW(lts, resonance) *
791:       PARAM_REGGE::ResonanceFormFactor(lts.m2, pow2(resonance.p.mass), resonance.g_FF);
792:
793:   double gammaflux1 = lts.excite1 ? IncohFlux(lts.x1, lts.t1, lts.qt1, lts.pfinal[1].M2())
794:                                   : CohFlux(lts.x1, lts.t1, lts.qt1);
795:   double gammaflux2 = lts.excite2 ? IncohFlux(lts.x2, lts.t2, lts.qt2, lts.pfinal[2].M2())
796:                                   : CohFlux(lts.x2, lts.t2, lts.qt2);
797:
798:   // "To amplitude level"
799:   gammaflux1 = msqrt(gammaflux1 / lts.x1);
800:   gammaflux2 = msqrt(gammaflux2 / lts.x2);
801:
802:   // Pomeron up (t1) x Photon down (t2)
803:   const std::complex<double> A_1 =
804:       gammaflux2 * common *
805:       PhotoProp(lts.s1, lts.t1, pow2(resonance.p.mass), lts.excite1, lts.pfinal[1].M2());
806:
807:   // Photon up (t1) x Pomeron down (t2)
808:   const std::complex<double> A_2 =
809:       gammaflux1 * common *
810:       PhotoProp(lts.s2, lts.t2, pow2(resonance.p.mass), lts.excite2, lts.pfinal[2].M2());
811:
812:   // Should sum here with negative sign if proton-antiproton initial state (anti-symmetric)
813:   const std::complex<double> A_prod = (lts.beam1.pdg == lts.beam2.pdg) ? (A_1 + A_2) : (A_1 - A_2);
814:
815:   // Production and Decay amplitude
816:   const std::complex<double> A_spin =
817:       spin::ProdAmp(lts, resonance) * spin::DecayAmp(lts, resonance);
818:
819:   // Full amplitude
820:   const std::complex<double> A = A_prod * A_spin;
821:
822:   // ------------------------------------------------------------------------
823:   // For screening loop
824:   lts.hamp = {A};
825:   // ------------------------------------------------------------------------
826:
827:   return A;
828: }
829:
830:
```



```
831: // ============================================================================
832: // Pomeron propagator x coupling to proton in photoproduction
833: //
834: // Essentially, parametrized Regge scaling of
835: // quasielastic photon + proton -> vectormeson + proton
836: //
837: // dsigma(vector meson)/dt ~ (W_yp^2/W0^2)^(2(alpha(t) - 1)) x exp(B_0 t),
838: //
839: // where the trajectory follows
840: // - soft pomeron for rho (deltaÅ ~ 0.08)
841: // - hard pomeron for j/psi (delta ~ 0.2) ... upsilon (delta ~ 0.3)
842: //
843: // HERA notation uses usually Q^2 for the lepton side momentum transfer
844: // and t for the proton (pomeron) side.
845: //
846: //
847: // [REFERENCE: Donnachie, Dosch, Landshoff, Nachtmann, Pomeron Physics and QCD, 2002]
848:
849: std::complex<double> MRegge::PhotoProp(double s, double t, double m2, bool excite,
850:                                        double M2_forward) const {
851:   // Different vector meson slopes (HERA data),
852:   // dsigma/dt ~ exp(Bt),
853:   // where B = B0 + 4alpha'log(W/W0) = B0 + 2alpha'log(W^2/W0^2)
854:   // W0 reference scale usually 90...95 GeV at which B0 is fitted from
855:   // data
856:   // B0 ~ 11 for rho and 4.8 for J/psi, Upsilon
857:   //
858:   // A running slope fit to vector meson / photoproduction slope data
859:   const double m_freeze = 0.7;  // GeV, Minimum mass freezing of running
860:   const double B0 =
861:       std::pow(std::max(pow2(m_freeze), m2) / 2.0, -gra::math::PI / 2.0) + 3 * gra::math::PI / 2.0;
862:
863:   // Normalization scale GeV^2
864:   // const double W02 = pow2(90.0); // Typical HERA data normalization
865:   const double W02 = pow2(1.0);  // We normalize to 1
866:
867:   // N.B. The pomeron slope (a') also affects, so all parameters need
868:   // to be fitted in a proper way (full MC)
869:   const double                alpha    = PARAM_REGGE::a0[0] + PARAM_REGGE::ap[0] * t;
870:   const double                alpha_t0 = PARAM_REGGE::a0[0];
871:   const double                ap       = PARAM_REGGE::ap[0];
872:   const std::complex<double> eta      = ReggeEtaLinear(t, alpha_t0, ap, PARAM_REGGE::sgn[0]);
873:
874:   // Proton form factor simply exponential here
875:   // Division by 2, because we are at amplitude level
876:
877:   // Proton / Dissociative system vertex
878:   const double FF =
879:       excite ? msqrt(PARAM_SOFT::g3P * PARAM_SOFT::gN_P) * gra::form::S3FINEL(t, M2_forward)
880:              : PARAM_SOFT::gN_P * std::exp(B0 * t / 2.0);
881:
882:   return eta * std::pow(s / W02, alpha) * FF;
883: }
884:
885: // ============================================================================
886: // Regge propagators in the form (s/s0)^alpha(t)
887: //
888: std::complex<double> MRegge::PropOnly(double s, double t) const {
889:   std::complex<double> M(0, 0);
890:
891:   // Loop over Pomeron, Reggeon, Reggeon... exchanges
892:   for (std::size_t k = 0; k < PARAM_REGGE::a0.size(); ++k) {
893:     if (PARAM_REGGE::n[k] == true) {  // Trajectory active
894:
895:       // Trajectory signature
896:       const double                alpha    = PARAM_REGGE::a0[k] + PARAM_REGGE::ap[k] * t;
897:       const double                alpha_t0 = PARAM_REGGE::a0[k];
898:       const double                ap       = PARAM_REGGE::ap[k];
899:       const std::complex<double> eta      = ReggeEtaLinear(t, alpha_t0, ap, PARAM_REGGE::sgn[k]);
900:
901:       // Add to the sum
902:       M += eta * std::pow(s / PARAM_REGGE::s0, alpha);
903:     }
904:   }
905:   return M;
906: }
907:
908: // ============================================================================
909: // Odderon propagators in the form (s/s0)^alpha(t)
910: //
911: std::complex<double> MRegge::OdderonProp(double s, double t) const {
912:   // Use Pomeron trajectory with odd signature
913:   const double                alpha    = PARAM_REGGE::a0[0] + PARAM_REGGE::ap[0] * t;
```



```
914:   const double                alpha_t0 = PARAM_REGGE::a0[0];
915:   const double                ap       = PARAM_REGGE::ap[0];
916:   const std::complex<double> eta      = ReggeEtaLinear(t, alpha_t0, ap, (-1) * PARAM_REGGE::sgn[0]);
917:
918:   return eta * std::pow(s / PARAM_REGGE::s0, alpha);
919: }
920:
921: // ------------------------------------------------------------------------
922:
923: // Regge amplitude parameters
924: namespace PARAM_REGGE {
925: bool initialized = false;
926:
927: std::vector<double> a0;
928: std::vector<double> ap;
929: std::vector<double> sgn;
930:
931: double s0 = 0.0;
932:
933: int    offshellFF = 0;
934: double b_EXP      = 0.0;
935: double a_OREAR    = 0.0;
936: double b_OREAR    = 0.0;
937: double b_POW      = 0.0;
938:
939: bool reggeize = false;
940:
941: double omega = 0.0;
942:
943: std::vector<double> c;  // coupling
944: std::vector<bool>   n;  // on/off
945:
946: void PrintParam() {
947:   std::cout << "PARAM_REGGE:: Sub-amplitude parameters:" << std::endl << std::endl;
948:   for (unsigned int i = 0; i < a0.size(); ++i) {
949:     printf(
950:         "- Reggeon[%d]: trajectory alpha(t) = %0.3f + %0.3f "
951:         "[GeV^{-2}] t, sgn = %0.0f \n",
952:         i, a0[i], ap[i], sgn[i]);
953:   }
954:   printf("- offshellFF = %d \n", offshellFF);
955:   if (offshellFF == 0) { printf("  -- b_EXP = %0.3f [GeV^{-2}] (exponential) \n", b_EXP); }
956:   if (offshellFF == 1) {
957:     printf(
958:         "  -- a_OREAR = %0.3f [GeV^{-1}], b_OREAR = %0.3f "
959:         "[GeV^{-1}] (Orear)\n",
960:         a_OREAR, b_OREAR);
961:   }
962:   if (offshellFF == 2) { printf("  -- b_POW = %0.3f [GeV^{-2}] (powerlaw) \n", b_POW); }
963:   printf("- reggeize = %d \n", reggeize);
964:   printf("- s0 = %0.3f [GeV^2]", s0);
965:   std::cout << std::endl << std::endl;
966:
967:   std::cout << "Couplings:" << std::endl;
968:   for (unsigned int i = 0; i < c.size(); ++i) {
969:     printf("- Reggeon[%d]: c = %0.3f [GeV^{-2}]", i, c[i]);
970:     if (n[i]) {
971:       std::cout << rang::fg::green << " [on]" << rang::fg::reset << std::endl;
972:     } else {
973:       std::cout << rang::fg::red << " [off]" << rang::fg::reset << std::endl;
974:     }
975:   }
976:   std::cout << std::endl;
977: }
978:
979: // Read generic Regge amplitude parameters from files
980: // PDG is the pion/kaon/proton
981: // TODO: we read couplings based on the first particle,
982: // generalize to pairs such as pi+pi-K+K- (4-body) (easy extension)
983: //
984: void ReadParameters(int PDG, const std::string &modelfile) {
985:   // Read and parse
986:   try {
987:     const std::string data = gra::aux::GetInputData(modelfile);
988:     nlohmann::json     j    = nlohmann::json::parse(data);
989:
990:     const std::string XID = "PARAM_REGGE";
991:
992:     // Regge amplitude parameters
993:     std::vector<double> a0  = j.at(XID).at("a0");
994:     std::vector<double> ap  = j.at(XID).at("ap");
995:     std::vector<double> sgn = j.at(XID).at("sgn");
996:     PARAM_REGGE::a0         = a0;
```



```
997:     PARAM_REGGE::ap         = ap;
998:     PARAM_REGGE::sgn        = sgn;
999:     PARAM_REGGE::s0         = j.at(XID).at("s0");
1000:
1001:     PARAM_REGGE::offshellFF = j.at(XID).at("offshellFF");
1002:     PARAM_REGGE::b_EXP      = j.at(XID).at("b_EXP");
1003:     PARAM_REGGE::a_OREAR    = j.at(XID).at("a_OREAR");
1004:     PARAM_REGGE::b_OREAR    = j.at(XID).at("b_OREAR");
1005:     PARAM_REGGE::b_POW      = j.at(XID).at("b_POW");
1006:     PARAM_REGGE::reggeize   = j.at(XID).at("reggeize");
1007:     PARAM_REGGE::omega      = j.at(XID).at("omega");
1008:
1009:     // Setup the parameter string
1010:     std::string str;
1011:     if (PDG == 211 || PDG == 111) {  // Charged or Neutral Pions
1012:       str = "PARAM_PI";
1013:     } else if (PDG == 321 || PDG == 311) {  // Charged or Neutral Kaons
1014:       str = "PARAM_K";
1015:     } else if (PDG == 2212 || PDG == 2112) {  // Protons or Neutrons
1016:       str = "PARAM_P";
1017:     } else if (PDG == 113 || PDG == 213) {  // Neutral or charged rho(770)
1018:       str = "PARAM_RHO";
1019:     } else if (PDG == 331) {  // phi(1020)
1020:       str = "PARAM_PHI";
1021:     } else if (PDG == 9000221) {  // f0(500)
1022:       str = "PARAM_F0500";
1023:     } else {  // The rest will use PARAM_X setup
1024:       str = "PARAM_X";
1025:     }
1026:
1027:     // Reggeon couplings
1028:     std::vector<double> c = j.at(str).at("c");
1029:     std::vector<bool>   n = j.at(str).at("n");
1030:     PARAM_REGGE::c        = c;
1031:     PARAM_REGGE::n        = n;
1032:
1033:     PARAM_REGGE::initialized = true;
1034:
1035:     // PARAM_REGGE::PrintParam();
1036:   } catch (...) {
1037:     throw std::invalid_argument("REGGE_PARAM::ReadParameters: Parse error in " + modelfile);
1038:   }
1039: }
1040:
1041: // Non-Conserved Vector Current Pomeron [UNTESTED FUNCTION]
1042: //
1043: // [REFERENCE: Close, Schuler, https://arxiv.org/pdf/hep-ph/9902243.pdf]
1044: // [REFERENCE: Close, Schuler, https://arxiv.org/pdf/hep-ph/9905305.pdf]
1045: //
1046: // see also:
1047: // [REFERENCE: Kaidalov, Khoze, Martin, Ryskin,
1048: // https://arxiv.org/pdf/hep-ph/0307064.pdf]
1049: //
1050: std::complex<double> JPC_CS_coupling(const gra::LORENTZSCALAR &lts,
1051:                                      const gra::PARAM_RES &    resonance) {
1052:   const unsigned int PP = 0;  // +1 +1
1053:   const unsigned int PM = 1;  // +1 -1
1054:   const unsigned int LL = 2;  //  0  0
1055:
1056:   const unsigned int PL = 3;  // +1  0
1057:   const unsigned int LP = 4;  //  0 +1
1058:
1059:   std::vector<double> A(5, 0.0);
1060:
1061:   // ------------------------------------------------------------------
1062:
1063:   const int    J   = resonance.p.spinX2 / 2.0;
1064:   const int    P   = resonance.p.P;
1065:   const double phi = lts.pfinal[1].DeltaPhi(lts.pfinal[2]);
1066:
1067:   // Dynamic structure (approx)
1068:   const double Q1_2  = std::abs(lts.t1);
1069:   const double Q2_2  = std::abs(lts.t2);
1070:   const double D     = (Q1_2 - Q2_2) / lts.m2;
1071:   const double delta = msqrt(Q1_2) * msqrt(Q2_2) / lts.m2;
1072:
1073:   // Naturality
1074:   const double eta1 = 1;                        // Pomeron 1 naturality (parity x signature)
1075:   const double eta2 = 1;                        // Pomeron 2 naturality (parity x signature)
1076:   const double etaM = P * std::pow(-1, J);  // Central boson naturality
1077:   const double eta  = eta1 * eta2 * etaM;
1078:
1079:   // ------------------------------------------------------------------
```

```
1080:
1081:   // Coupling structure (capture local variables with &)
1082:   auto Q = [&]() { return lts.t1 * lts.t2 * pow2(A[PP]) * pow2(std::sin(phi)); };
1083:   auto W = [&]() {
1084:     const double xi1 = gra::math::csgn(A[PP]) * gra::math::csgn(A[LL]);
1085:     return pow2(msqrt(lts.t1 * lts.t2) * A[LL] -
1086:                 xi1 * msqrt(lts.t1 * lts.t2) * A[PP] * std::cos(phi));
1087:   };
1088:   auto E = [&]() {
1089:     const double xi2   = gra::math::csgn(A[PL]) * gra::math::csgn(A[LP]);
1090:     const double alpha = lts.qt2 * A[PL];
1091:     const double beta  = eta * xi2 * lts.qt1 * A[LP];
1092:     return (lts.q1 * alpha - lts.q2 * beta).Pt2();
1093:   };
1094:   auto R = [&]() { return lts.t1 * lts.t2 * 0.5 * pow2(A[PM]); };
1095:
1096:   // ------------------------------------------------------------------
1097:
1098:   const double nabla = 0;
1099:
1100:   // Selection rules
1101:   if (J == 0 && P == -1) {
1102:     A[LL] = 0;
1103:     A[PP] = 1;
1104:     A[PM] = 0;
1105:
1106:     return msqrt(Q());
1107:   }
1108:   if (J == 0 && P == 1) {
1109:     A[LL] = nabla * delta;
1110:     A[PP] = 1;
1111:     A[PM] = 0;
1112:     return msqrt(W());
1113:   }
1114:   if (J == 1 && P == -1) {
1115:     A[LL] = nabla * D * delta;
1116:     A[PP] = D;
1117:     A[PM] = 0;
1118:
1119:     return msqrt(Q() + W());
1120:   }
1121:   if (J == 1 && P == 1) {
1122:     A[LL] = 0;
1123:     A[PP] = D;
1124:     A[PM] = 0;
1125:
1126:     return msqrt(Q() + W());
1127:   }
1128:   if (J == 2 && P == -1) {
1129:     A[LL] = 0;
1130:     A[PP] = 1;
1131:     A[PM] = D;
1132:     A[PL] = msqrt(Q2_2 / lts.m2);
1133:     A[LP] = msqrt(Q1_2 / lts.m2);
1134:
1135:     return msqrt(Q() + E() + R());
1136:   }
1137:   if (J == 2 && P == 1) {
1138:     A[LL] = nabla * delta;
1139:     A[PP] = 1;
1140:     A[PM] = 1;
1141:     A[PL] = msqrt(Q2_2 / lts.m2);
1142:     A[LP] = msqrt(Q1_2 / lts.m2);
1143:
1144:     return msqrt(W() + E() + R());
1145:   }
1146:
1147:   return 1.0;
1148: }
1149:
1150: // For data, see:
1151: //
1152: // [REFERENCE: Kirk for WA102, https://arxiv.org/abs/hep-ph/9908253v1]
1153:
1154: // Off-Shell meson form factor
1155: double Meson_FF(double t_hat, double M2) {
1156:   const double tprime = t_hat - M2;
1157:
1158:   switch (PARAM_REGGE::offshellFF) {
1159:     // Exponential
1160:     case 0:
1161:       return std::exp(-PARAM_REGGE::b_EXP * std::abs(tprime));
1162:     // Orear
```



```
1163:     case 1:
1164:       return std::exp(-PARAM_REGGE::b_OREAR * msqrt(std::abs(tprime) + pow2(PARAM_REGGE::a_OREAR)) +
1165:                       PARAM_REGGE::a_OREAR * PARAM_REGGE::b_OREAR);
1166:     // Powerlaw
1167:     case 2:
1168:       return 1.0 / (1.0 - tprime / PARAM_REGGE::b_POW);
1169:
1170:     default:
1171:       throw std::invalid_argument("Meson_FF: Unknown parameter: " +
1172:                                   std::to_string(PARAM_REGGE::offshellFF));
1173:   }
1174: }
1175:
1176: // Off-Shell baryon form factor
1177: double Baryon_FF(double t_hat, double M2) {
1178:   const double tprime = t_hat - M2;
1179:
1180:   switch (PARAM_REGGE::offshellFF) {
1181:     // Exponential
1182:     case 0:
1183:       return std::exp(-PARAM_REGGE::b_EXP * std::abs(tprime));
1184:     // Orear
1185:     case 1:
1186:       return std::exp(-PARAM_REGGE::b_OREAR * msqrt(std::abs(tprime) + pow2(PARAM_REGGE::a_OREAR)) +
1187:                       PARAM_REGGE::a_OREAR * PARAM_REGGE::b_OREAR);
1188:     // Powerlaw
1189:     case 2:
1190:       return 1.0 / (1.0 - tprime / PARAM_REGGE::b_POW);
1191:
1192:     default:
1193:       throw std::invalid_argument("Baryon_FF: Unknown parameter: " +
1194:                                   std::to_string(PARAM_REGGE::offshellFF));
1195:   }
1196: }
1197:
1198: // Off-shell meson M* propagator
1199: double Meson_prop(double t_hat, double M2) {
1200:   double val = 1.0 / (t_hat - M2);
1201:   if (PARAM_REGGE::reggeize) {
1202:     //"Reggeization"  s^J(t) (NOT IMPLEMENTED, return without)
1203:     return val;
1204:   } else {
1205:     return val;
1206:   }
1207: }
1208:
1209: // Off-shell baryon B* propagator
1210: double Baryon_prop(double t_hat, double M2) {
1211:   double val = 1.0 / (t_hat - M2);
1212:   if (PARAM_REGGE::reggeize) {
1213:     //"Reggeization"  s^J(t) (NOT IMPLEMENTED, return without)
1214:     return val;
1215:   } else {
1216:     return val;
1217:   }
1218: }
1219:
1220: // Final State Interaction (FSI) kt^2 loop integral
1221: // NOT IMPLEMENTED
1222: std::complex<double> FSI_prop(double t_hat, double M2) {
1223:   std::complex<double> value(1, 0);
1224:   return value;
1225: }
1226:
1227: // Resonance form factor
1228: // (in addition to Breit-Wigner propagator)
1229: //
1230: // In general, without any interfering continuum process
1231: // (which would provide the same effect), this may be needed?
1232: //
1233: double ResonanceFormFactor(double s_hat, double M2, double S0) {
1234:   return std::exp(-pow2(s_hat - M2) / math::pow4(S0));
1235: }
1236:
1237: }  // namespace PARAM_REGGE
1238: }  // namespace gra
1239:
```



```
  1: // 2D histogram class
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: // C++
  7: #include <complex>
  8: #include <iostream>
  9: #include <vector>
 10:
 11: // Own
 12: #include "Granitti/MAux.h"
 13: #include "Granitti/MH2.h"
 14: #include "Granitti/MMath.h"
 15: #include "Granitti/MMatrix.h"
 16:
 17: // Libraries
 18: #include "rang.hpp"
 19:
 20: namespace gra {
 21: // Constructor
 22: MH2::MH2(int xbins, double xmin, double xmax, int ybins, double ymin, double ymax,
 23:          std::string namestr) {
 24:   name = namestr;
 25:   ResetBounds(xbins, xmin, xmax, ybins, ymin, ymax);
 26:   FILLBUFF = false;
 27: }
 28:
 29: // Constructor with only number of bins
 30: MH2::MH2(int xbins, int ybins, std::string namestr) {
 31:   name     = namestr;
 32:   XBINS    = xbins;
 33:   YBINS    = ybins;
 34:   FILLBUFF = true;
 35: }
 36:
 37: // Empty constructor
 38: MH2::MH2() {
 39:   XBINS    = 50;  // Default
 40:   YBINS    = 50;
 41:   FILLBUFF = true;
 42: }
 43:
 44: // Destructor
 45: MH2::~MH2() {}
 46:
 47: void MH2::ResetBounds(int xbins, double xmin, double xmax, int ybins, double ymin, double ymax) {
 48:   XMIN  = xmin;
 49:   XMAX  = xmax;
 50:   XBINS = xbins;
 51:
 52:   YMIN  = ymin;
 53:   YMAX  = ymax;
 54:   YBINS = ybins;
 55:
 56:   // Init
 57:   weights  = MMatrix<double>(XBINS, YBINS, 0.0);
 58:   weights2 = weights;
 59:   counts   = MMatrix<long long int>(XBINS, YBINS, 0);
 60:
 61:   Clear();           // Call also this!
 62:   FILLBUFF = false;  // No autorange, explicit bounds provided
 63: }
 64:
 65: void MH2::Print() const {
 66:   if (!(fills > 0)) {  // No fills
 67:     std::cout << "MH2::Print: <" << name << "> Fills = " << fills << std::endl;
 68:     return;
 69:   }
 70:
 71:   // Histogram name
 72:   std::cout << "MH2::Print: <" << name << ">" << std::endl;
 73:
 74:   std::vector<std::string> ascziart = {" ", ".", ":", "-", "=", "+", "*", "#", "%", "@"};
 75:   const double             maxw     = GetMaxWeight();
 76:
 77:   std::cout << "          |";  // Empty top left corner
 78:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) { std::cout << "="; }
 79:   std::cout << "| " << std::endl;
 80:
 81:   for (int j = YBINS - 1; j > -1; --j) {  // Must be int, we index down to negative
 82:     const double binwidth = (YMAX - YMIN) / YBINS;
 83:     printf("%9.2E |", binwidth * (j + 1) - binwidth / 2.0 + YMIN);
```



```
 84:     for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
 85:       const double w   = (maxw > 0) ? weights[i][j] / maxw : 0;
 86:       const int    ind = std::round(w * 9);
 87:       std::cout << rang::fg::yellow << ascziart[ind] << rang::fg::reset;
 88:     }
 89:     std::cout << "|";
 90:     std::cout << std::endl;
 91:   }
 92:   std::cout << "          |";  // Empty bottom left corner
 93:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) { std::cout << "="; }
 94:   std::cout << "| " << std::endl;
 95:
 96:   const double binwidth = (XMAX - XMIN) / XBINS;
 97:   for (int k = -1; k < 50; ++k) {
 98:     std::cout << "           ";
 99:     int empty = 0;
100:     for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
101:       const double      binvalue = binwidth * (i + 1) - binwidth / 2.0 + XMIN;
102:       const std::string let      = gra::aux::ToString(std::abs(binvalue), 2);
103:       const std::string sgn      = (binvalue >= 0.0) ? "+" : "-";
104:       if (k == -1) {  // print value sign (-+)
105:         std::cout << sgn;
106:         continue;
107:       }
108:       if (k < static_cast<int>(let.length())) {
109:         std::cout << let[k];
110:       } else {
111:         std::cout << " ";
112:         empty++;
113:       }
114:     }
115:     std::cout << std::endl;
116:     if (empty == XBINS) { break; }  // the end
117:   }
118:
119:   // Print statistics
120:   std::cout << "<binned statistics> " << std::endl;
121:   std::pair<double, double> valerr = WeightMeanAndError();
122:   printf(" <W> = %0.3E +- %0.3E  [# = %lld | U: U/O = %lld/%lld, V: U/O = %lld/%lld ]\n",
123:          valerr.first, valerr.second, fills, underflow[0], overflow[0], underflow[1], overflow[1]);
124:
125:   const double X_mean  = GetMeanX(1);
126:   const double X_sgmean = GetMeanX(2);
127:
128:   const double Y_mean  = GetMeanY(1);
129:   const double Y_sgmean = GetMeanY(2);
130:
131:   printf(" <X> = %0.3f, <X^2> = %0.3f, <X^2> - <X>^2 = %0.3f \n", X_mean, X_sgmean,
132:          X_sgmean - std::pow(X_mean, 2));
133:   printf(" <Y> = %0.3f, <Y^2> = %0.3f, <Y^2> - <Y>^2 = %0.3f \n", Y_mean, Y_sgmean,
134:          Y_sgmean - std::pow(Y_mean, 2));
135:
136:   std::cout << std::endl;
137: }
138:
139: std::pair<double, double> MH2::WeightMeanAndError() const {
140:   const double N    = fills;  // Need to use number of total fills here, not counts in bins
141:   const double val  = SumWeights() / N;
142:   const double err2 = SumWeights2() / N - gra::math::pow2(val);
143:   const double err  = gra::math::msqrt(err2 / N);
144:
145:   return {val, err};
146: }
147:
148: bool MH2::ValidBin(int xbin, int ybin) const {
149:   if (xbin >= 0 && xbin < XBINS && ybin >= 0 && ybin < YBINS) { return true; }
150:   return false;
151: }
152:
153: void MH2::Fill(double xvalue, double yvalue) {
154:   // Call weighted fill with weight 1.0
155:   Fill(xvalue, yvalue, 1.0);
156: }
157:
158: void MH2::Fill(double xvalue, double yvalue, double weight) {
159:   if (std::isnan(weight)) {
160:     std::cout << "MH2:Fill: Warning: NaN weight" << std::endl;
161:     return;
162:   }
163:   if (std::isinf(weight)) {
164:     std::cout << "MH2:Fill: Warning: Inf weight" << std::endl;
165:     return;
166:   }
```

```
167:
168:   if (!FILLBUFF) {  // Normal filling
169:
170:     ++fills;
171:
172:     // Find out bins
173:     const int xbin = GetIdx(xvalue, XMIN, XMAX, XBINS, LOGX);
174:     const int ybin = GetIdx(yvalue, YMIN, YMAX, YBINS, LOGY);
175:
176:     if (xbin == -3 || ybin == -3) { nanflow += 1; }
177:
178:     if (xbin == -1) { underflow[0] += 1; }
179:     if (ybin == -1) { underflow[1] += 1; }
180:
181:     if (xbin == -2) { overflow[0] += 1; }
182:     if (ybin == -2) { overflow[1] += 1; }
183:
184:     if (ValidBin(xbin, ybin)) {
185:       weights[xbin][ybin] += weight;
186:       weights2[xbin][ybin] += weight * weight;
187:       counts[xbin][ybin] += 1;
188:     }
189:   } else {  // Autorange initialization
190:
191:     buff_values.push_back({xvalue, yvalue});
192:     buff_weights.push_back(weight);
193:
194:     if (buff_values.size() > static_cast<unsigned int>(AUTOBUFFSIZE)) { FlushBuffer(); }
195:   }
196: }
197:
198: void MH2::Clear() {
199:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
200:     for (std::size_t j = 0; j < static_cast<unsigned int>(YBINS); ++j) {
201:       weights[i][j]  = 0;
202:       weights2[i][j] = 0;
203:       counts[i][j]   = 0;
204:     }
205:   }
206:   fills     = 0;
207:   underflow = {0, 0};
208:   overflow  = {0, 0};
209:   nanflow   = 0;
210: }
211:
212:
213: // Automatic histogram range algorithm
214: void MH2::FlushBuffer() {
215:   if (FILLBUFF && buff_values.size() > 0) {
216:     FILLBUFF = false;  // no more filling buffer
217:
218:     std::vector<double> min = {0.0, 0.0};
219:     std::vector<double> max = {0.0, 0.0};
220:
221:     // Loop over dimensions
222:     for (std::size_t dim = 0; dim < 2; ++dim) {
223:       // Find out mean
224:       double mu   = 0;
225:       double sumW = 0;
226:       for (std::size_t i = 0; i < buff_values.size(); ++i) {
227:         mu += buff_values[i][dim] * buff_weights[i];
228:         sumW += buff_weights[i];
229:       }
230:       if (sumW > 0) { mu /= sumW; }
231:
232:       // Variance
233:       double var = 0;
234:       for (std::size_t i = 0; i < buff_values.size(); ++i) {
235:         var += buff_weights[i] * std::pow(buff_values[i][dim] - mu, 2);
236:       }
237:       if (sumW > 0) { var /= sumW; }
238:
239:       // Find minimum and maximum
240:       double minval = 1e64;
241:       double maxval = -1e64;
242:       for (std::size_t i = 0; i < buff_values.size(); ++i) {
243:         if (buff_values[i][dim] < minval) { minval = buff_values[i][dim]; }
244:         if (buff_values[i][dim] > maxval) { maxval = buff_values[i][dim]; }
245:       }
246:
247:       // Set new histogram bounds
248:       double std  = std::sqrt(std::abs(var));
249:       double xmin = mu - 2.5 * std;
```



```
250:       double xmax = mu + 2.5 * std;
251:
252:       // A numerical failure may happen with variance calculation, then use this
253:       if (std::isnan(xmin) || std::isnan(xmax)) {
254:         xmin = minval;
255:         xmax = maxval;
256:       }
257:
258:       // If symmetric setup set by user
259:       if (AUTOSYMMETRY[dim]) {
260:         double val = (std::abs(xmin) + std::abs(xmax)) / 2.0;
261:         xmin       = -val;
262:         xmax       = val;
263:       }
264:
265:       // We have only positive values, such as invariant mass
266:       if (minval > 0.0) { xmin = std::max(0.0, xmin); }
267:
268:       min[dim] = xmin;
269:       max[dim] = xmax;
270:     }
271:
272:     // New histogram bounds
273:     ResetBounds(XBINS, min[0], max[0], YBINS, min[1], max[1]);
274:
275:     // Fill buffered events
276:     for (std::size_t i = 0; i < buff_values.size(); ++i) {
277:       Fill(buff_values[i][0], buff_values[i][1], buff_weights[i]);
278:     }
279:
280:     // Clear buffers
281:     buff_values.clear();
282:     buff_weights.clear();
283:   }
284: }
285:
286: // Histogram mean (based on binned values)
287: double MH2::GetMeanX(int power) const {
288:   double       sum      = 0.0;
289:   double       norm     = 0.0;  // Normalization
290:   const double binwidth = (XMAX - XMIN) / XBINS;
291:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
292:     const double value = std::pow(binwidth * (i + 1) - binwidth / 2.0 + XMIN, power);
293:     for (std::size_t j = 0; j < static_cast<unsigned int>(YBINS); ++j) {
294:       const double weight = GetBinWeight(i, j);
295:       sum += weight * value;
296:       norm += weight;
297:     }
298:   }
299:   if (norm > 0) {
300:     return sum / norm;
301:   } else {
302:     return 0.0;
303:   }
304: }
305: double MH2::GetMeanY(int power) const {
306:   double       sum      = 0.0;
307:   double       norm     = 0.0;  // Normalization
308:   const double binwidth = (YMAX - YMIN) / YBINS;
309:   for (std::size_t j = 0; j < static_cast<unsigned int>(YBINS); ++j) {
310:     const double value = std::pow(binwidth * (j + 1) - binwidth / 2.0 + YMIN, power);
311:     for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
312:       const double weight = GetBinWeight(i, j);
313:       sum += weight * value;
314:       norm += weight;
315:     }
316:   }
317:   if (norm > 0) {
318:     return sum / norm;
319:   } else {
320:     return 0.0;
321:   }
322: }
323:
324:
325: double MH2::SumWeights() const {
326:   double sum = 0.0;
327:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
328:     for (std::size_t j = 0; j < static_cast<unsigned int>(YBINS); ++j) { sum += weights[i][j]; }
329:   }
330:   return sum;
331: }
332:
```



```
333: double MH2::SumWeights2() const {
334:   double sum = 0.0;
335:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
336:     for (std::size_t j = 0; j < static_cast<unsigned int>(YBINS); ++j) { sum += weights2[i][j]; }
337:   }
338:   return sum;
339: }
340:
341: long long int MH2::SumBinCounts() const {
342:   long long int sum = 0;
343:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
344:     for (std::size_t j = 0; j < static_cast<unsigned int>(YBINS); ++j) { sum += counts[i][j]; }
345:   }
346:   return sum;
347: }
348:
349: double MH2::GetMaxWeight() const {
350:   double maxval = 0;
351:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
352:     for (std::size_t j = 0; j < static_cast<unsigned int>(YBINS); ++j) {
353:       if (weights[i][j] > maxval) { maxval = weights[i][j]; }
354:     }
355:   }
356:   return maxval;
357: }
358:
359: double MH2::GetMinWeight() const {
360:   double minval = 1e128;
361:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
362:     for (std::size_t j = 0; j < static_cast<unsigned int>(YBINS); ++j) {
363:       if (weights[i][j] < minval) { minval = weights[i][j]; }
364:     }
365:   }
366:   return minval;
367: }
368:
369: long long int MH2::GetBinCount(int xbin, int ybin) const {
370:   if (ValidBin(xbin, ybin)) {
371:     return counts[xbin][ybin];
372:   } else {
373:     return 0;
374:   }
375: }
376:
377: // Get weight of the bin
378: double MH2::GetBinWeight(int xbin, int ybin) const {
379:   if (ValidBin(xbin, ybin)) {
380:     return weights[xbin][ybin];
381:   } else {
382:     return 0;
383:   }
384: }
385:
386: // Get bin indices (i,j) corresponding to value (xvalue,yvalue)
387: void MH2::GetBinIdx(double xvalue, double yvalue, int &xbin, int &ybin) const {
388:   // Find out bins
389:   xbin = GetIdx(xvalue, XMIN, XMAX, XBINS, LOGX);
390:   ybin = GetIdx(yvalue, YMIN, YMAX, YBINS, LOGY);
391: }
392:
393: // Get table/histogram index for linearly or base-10 logarithmically spaced
394: // bins
395: // Gives exact uniform filling within bin boundaries.
396: //
397: // In the logarithmic case, MINVAL and MAXVAL > 0, naturally.
398: //
399: //
400: // Underflow returns -1
401: // Overflow  returns -2
402: // nan/inf   returns -3
403: //
404: int MH2::GetIdx(double value, double minval, double maxval, int nbins, bool logbins) const {
405:   if (std::isnan(value) || std::isinf(value)) { return -3; }
406:   if (value < minval) { return -1; }  // Underflow
407:   if (value > maxval) { return -2; }  // Overflow
408:   int idx = 0;
409:
410:   // Logarithmic binning
411:   if (logbins) {
412:     // Check do we have a non-negative input
413:     idx = value > 0 ? std::floor(nbins * (std::log10(value) - std::log10(minval)) /
414:                                  (std::log10(maxval) - std::log10(minval)))
415:                     : -1;
```



```
416:     // Linear binning
417:   } else {
418:     const double BINWIDTH = (maxval - minval) / nbins;
419:     idx                   = std::floor((value - minval) / BINWIDTH);
420:   }
421:   return idx;
422: }
423:
424: // Return Shannon entropy of the histogram in bits (base-2 log)
425: // (useful for validating statistical properties, for example)
426: double MH2::ShannonEntropy() const {
427:   double sum = 0.0;
428:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
429:     for (std::size_t j = 0; j < static_cast<unsigned int>(YBINS); ++j) { sum += weights[i][j]; }
430:   }
431:   if (sum > 0) {
432:     double S = 0.0;
433:     for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
434:       for (std::size_t j = 0; j < static_cast<unsigned int>(YBINS); ++j) {
435:         if (weights[i][j] > 0) { S += weights[i][j] / sum * std::log2(weights[i][j] / sum); }
436:       }
437:     }
438:     return -S;
439:   } else {
440:     return 0.0;
441:   }
442: }
443:
444: }  // namespace gra
445:
```

```cpp
// 'Continuum' i.e. directly constructed phase space class
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

// C++
#include <algorithm>
#include <complex>
#include <iostream>
#include <random>
#include <vector>

// OWN
#include "Granititi/MAux.h"
#include "Granititi/MContinuum.h"
#include "Granititi/MForm.h"
#include "Granititi/MFragment.h"
#include "Granititi/MKinematics.h"
#include "Granititi/MMath.h"
#include "Granititi/MMatrix.h"
#include "Granititi/MProcess.h"
#include "Granititi/MSpin.h"
#include "Granititi/MUserCuts.h"

// Libraries
#include "rang.hpp"

using gra::PDG::GeV2barn;
using gra::aux::indices;
using gra::math::PI;
using gra::math::abs2;
using gra::math::msqrt;
using gra::math::pow2;
using gra::math::pow3;
using gra::math::pow4;
using gra::math::zi;

namespace gra {
// This is needed by construction
MContinuum::MContinuum() { Initialize(); }

// Constructor
MContinuum::MContinuum(std::string process, const std::vector<aux::OneCMD> &syntax) {
  Initialize();
  InitHistograms();
  SetProcess(process, syntax);

  // Init final states
  M4Vec zerovec(0, 0, 0, 0);
  for (std::size_t i = 0; i < 10; ++i) { lts.pfinal.push_back(zerovec); }
  std::cout << "MContinuum:: [Constructor done]" << std::endl;
}

void MContinuum::Initialize() {
  const std::vector<std::string> supported = {"PP", "yP", "OP", "yy", "gg"};
  CID                                      = "C";
  ProcPtr                                  = MSubProc(supported, CID);
}

// Destructor
MContinuum::~MContinuum() {}

// Initialize cut and process spesific postsetup
void MContinuum::post_Constructor() {
  std::cout << "MContinuum::post_Constructor: " << EXCITATION << std::endl;

  // Set sampling boundaries
  SetTechnicalBoundaries(gcuts, EXCITATION);

  // Technical cuts here
  if (gcuts.kt_max < 0.0) {  // not set by user

    gcuts.kt_min = 0.0;

    // Pomeron-Pomeron continuum (low-pt guaranteed by form factors)
    gcuts.kt_max = 10.0 / lts.decaytree.size();

    // QED/EW/Durham-QCD processes (high pt)
    if (ProcPtr.ISTATE != "PP" && ProcPtr.ISTATE != "yP" && ProcPtr.ISTATE != "OP") {
      gcuts.kt_max = lts.sqrt_s / 2.0;  // kinematic maximum
    }
  }

```



```cpp
  // Initialize phase space dimension (3*Nf - 4)
  ProcPtr.LIPSDIM = 3 * (lts.decaytree.size() + 2) - 4;

  if (EXCITATION == 1) { ProcPtr.LIPSDIM += 1; }
  if (EXCITATION == 2) { ProcPtr.LIPSDIM += 2; }
}

// Update kinematics (screening kT loop calls this)
// Exact 4-momentum conservation at loop vertices, that is, by using this one
// does not assume vanishing external momenta in the screening loop calculation
bool MContinuum::LoopKinematics(const std::vector<double> &p1p, const std::vector<double> &p2p) {
  static const M4Vec beamsum = lts.pbeam1 + lts.pbeam2;
  const unsigned int Nf      = lts.decaytree.size();  // Number of central particles
  const unsigned int Nf      = Kf + 2;                // Number of final states

  // SET new proton pT degrees of freedom
  lts.pfinal[1].SetPxPy(p1p[0], p1p[1]);
  lts.pfinal[2].SetPxPy(p2p[0], p2p[1]);

  // Get central final states pT degrees of freedom
  std::vector<M4Vec> p(Kf, M4Vec(0, 0, 0, 0));
  BLinearSystem(p, pkt_, lts.pfinal[1], lts.pfinal[2]);

  // Set central particles px,py,pz,e
  const unsigned int offset = 3;  // indexing
  M4Vec              sumP(0, 0, 0, 0);
  for (std::size_t i = 0; i < Kf; ++i) {
    const double y = lts.pfinal_orig[i + offset].Rap();
    const double m = lts.pfinal_orig[i + offset].M();

    // px,py
    lts.pfinal[i + offset].SetPxPy(p[i].Px(), p[i].Py());

    // pz,E
    const double mt = msqrt(pow2(m) + lts.pfinal[i + offset].Pt2());
    lts.pfinal[i + offset].SetPzE(mt * std::sinh(y), mt * std::cosh(y));

    sumP += lts.pfinal[i + offset];
  }

  // pz and E of forward protons/N*
  const double pt1 = lts.pfinal[1].Pt();
  const double pt2 = lts.pfinal[2].Pt();
  const double m1  = lts.pfinal_orig[1].M();
  const double m2  = lts.pfinal_orig[2].M();

  double p1z = gra::kinematics::SolvePz(m1, m2, pt1, pt2, sumP.Pz(), sumP.E(), lts.s);
  double p2z = -(sumP.Pz() + p1z);  // by momentum conservation

  // Enforce scattering direction +p -> +p, -p -> -p (VERY RARE POLYNOMIAL
  // BRANCH FLIP)
  if (p1z < 0 || p2z > 0) { return false; }

  lts.pfinal[1].SetPzE(p1z, msqrt(pow2(m1) + pow2(pt1) + pow2(p1z)));
  lts.pfinal[2].SetPzE(p2z, msqrt(pow2(m2) + pow2(pt2) + pow2(p2z)));
  lts.pfinal[0] = sumP;

  // ------------------------------------------------------------------
  // Now boost if asymmetric beams
  if (std::abs(beamsum.Pz()) > 1e-6) {
    constexpr int sign = 1;  // positive -> boost to the lab
    kinematics::LorentzBoost(beamsum, lts.sqrt_s, lts.pfinal[1], sign);
    kinematics::LorentzBoost(beamsum, lts.sqrt_s, lts.pfinal[2], sign);
    kinematics::LorentzBoost(beamsum, lts.sqrt_s, lts.pfinal[0], sign);

    for (std::size_t i = 0; i < Kf; ++i) {
      kinematics::LorentzBoost(beamsum, lts.sqrt_s, lts.pfinal[i + offset], sign);
    }
  }
  // ------------------------------------------------------------------

  // Check Energy-Momentum
  if (!gra::math::CheckEMC(beamsum - (lts.pfinal[1] + lts.pfinal[2] + lts.pfinal[0]))) {
    return false;
  }

  return GetLorentzScalars(Nf);
}

// Get weight
double MContinuum::EventWeight(const std::vector<double> &randvec, AuxIntData &aux) {
  double W = 0.0;

```



```cpp
  // Kinematics
  aux.kinematics_ok = BNRandomKin(lts.decaytree.size() + 2, randvec);
  aux.fiducts_ok    = FiducialCuts();
  aux.vetocuts_ok   = VetoCuts();

  if (aux.Valid()) {
    // Matrix element squared
    const double MatESQ = GetAmp2();

    // ------------------------------------------------------------------
    // Cascade resonances phase-space
    const double C_space = CascadePS();
    // ------------------------------------------------------------------

    // ** EVENT WEIGHT **
    W = C_space * (1.0 / S_factor) * BNPhaseSpaceWeight() * BNIntegralVolume() * MatESQ * GeV2barn /
        MollerFlux();
  }

  aux.amplitude_ok = CheckInfNan(W);

  // As the last step: Histograms
  const double totalweight = W * aux.vegasweight;
  FillHistograms(totalweight, lts);

  return W;
}

// Fiducial cuts
bool MContinuum::FiducialCuts() const { return CommonCuts(); }

// Record event
bool MContinuum::EventRecord(HepMC3::GenEvent &evt) { return CommonRecord(evt); }

void MContinuum::PrintInit(bool silent) const {
  if (!silent) {
    PrintSetup();

    // Construct prettyprint diagram
    std::string proton1 = "----------EL---------->";
    std::string proton2 = "----------EL---------->";

    if (EXCITATION == 1) { proton1 = "----------F2-xxxxxxxx>"; }
    if (EXCITATION == 2) {
      proton1 = "----------F2-xxxxxxxx>";
      proton2 = "----------F2-xxxxxxxx>";
    }

    std::string pomeronloop = (SCREENING == true) ? "     **    " : "           ";

    std::vector<std::string> legs;
    for (const auto &i : indices(lts.decaytree)) {
      char buff[250];
      snprintf(buff, sizeof(buff), "x---------> %d (%s) [Q=%s, J=%s]", lts.decaytree[i].p.pdg,
               lts.decaytree[i].p.name.c_str(),
               gra::aux::ChargeXtoString(lts.decaytree[i].p.chargeX3).c_str(),
               gra::aux::Spin2XtoString(lts.decaytree[i].p.spinX2).c_str());
      std::string leg = buff;
      legs.push_back(leg);
    }

    std::vector<std::string> feynmangraph;
    feynmangraph.push_back("||          ");
    for (const auto &i : indices(lts.decaytree)) {
      feynmangraph.push_back(legs[i]);
      if (i < lts.decaytree.size() - 1) feynmangraph.push_back("|          ");
    }
    feynmangraph.push_back("||          ");

    // Print diagram
    std::cout << proton1 << std::endl;
    for (const auto &i : indices(feynmangraph)) {
      if (SCREENING) {  // Put red
        std::cout << rang::fg::red << pomeronloop << rang::style::reset;
      } else {
        std::cout << rang::fg::red << pomeronloop << rang::style::reset;
      }
      std::cout << feynmangraph[i] << std::endl;
    }
    std::cout << proton2 << std::endl;
    std::cout << std::endl;

    // Generation cuts
```



```cpp
    std::cout << std::endl;
    std::cout << rang::style::bold << "Generation cuts:" << rang::style::reset << std::endl
              << std::endl;
    printf(
        "- Rap : Final state rapidity [min, max] = [%0.2f, %0.2f]     "
        "\t(user)     \n"
        "- Kt  : Intermediate        [min, max] = [%0.2f, %0.2f] GeV "
        "\t(fixed/user) \n"
        "- Pt  : Forward leg         [min, max] = [%0.2f, %0.2f] GeV "
        "\t(fixed/user) "
        "\n",
        gcuts.rap_min, gcuts.rap_max, gcuts.kt_min, gcuts.kt_max, gcuts.forward_pt_min,
        gcuts.forward_pt_max);

    if (EXCITATION != 0) {
      printf(
          "- Xi  : Forward leg (M^2/s)  [min, max] = [%0.2E, %0.2E]     "
          "\t(fixed/user) \n",
          gcuts.XI_min, gcuts.XI_max);
    }

    PrintFiducialCuts();
  }
}

// (3*Nf-4)-dimensional phase space vector initialization
bool MContinuum::BNRandomKin(unsigned int Nf, const std::vector<double> &randvec) {
  const unsigned int Kf = Nf - 2;  // Central system multiplicity

  // log-change of variables for pt
  const double u1 =
      std::log(gcuts.forward_pt_min + ZERO_EPS) +
      (std::log(gcuts.forward_pt_max) - std::log(gcuts.forward_pt_min + ZERO_EPS)) * randvec[0];
  const double u2 =
      std::log(gcuts.forward_pt_min + ZERO_EPS) +
      (std::log(gcuts.forward_pt_max) - std::log(gcuts.forward_pt_min + ZERO_EPS)) * randvec[1];

  const double pt1 = std::exp(u1);
  const double pt2 = std::exp(u2);

  const double       phi1   = 2.0 * gra::math::PI * randvec[2];
  const double       phi2   = 2.0 * gra::math::PI * randvec[3];
  const unsigned int offset = 4;  // 4 variables above

  // Decay product masses
  // ====================================================
  for (const auto &i : indices(lts.decaytree)) {
    GetOffShellMass(lts.decaytree[i], lts.decaytree[i].m_offshell);
  }
  // ====================================================

  // Intermediate kt
  std::vector<double> kt(Kf - 1, 0.0);  // Kf-1
  size_t              ind = offset;
  for (const auto &i : indices(kt)) {
    kt[i] = gcuts.kt_min + (gcuts.kt_max - gcuts.kt_min) * randvec[ind];
    ++ind;
  }

  // Intermediate phi
  std::vector<double> phi(Kf - 1, 0.0);  // Kf-1
  for (const auto &i : indices(phi)) {
    phi[i] = 2.0 * PI * randvec[ind];
    ++ind;
  }

  // Final state rapidity
  std::vector<double> y(Kf, 0.0);  // Kf
  for (const auto &i : indices(y)) {
    y[i] = gcuts.rap_min + (gcuts.rap_max - gcuts.rap_min) * randvec[ind];
    ++ind;
  }

  // Forward N* system masses
  std::vector<double> mvec;
  std::vector<double> rvec;
  if (EXCITATION == 1) { rvec = {randvec[ind]}; }
  if (EXCITATION == 2) { rvec = {randvec[ind], randvec[ind + 1]}; }
  SampleForwardMasses(mvec, rvec);

  return BNBuildKin(Nf, pt1, pt2, phi1, phi2, kt, phi, y, mvec[0], mvec[1]);
}
```

```
333: // Build kinematics of 2->N
334: bool MContinuum::BNBuildKin(unsigned int Nf, double pt1, double pt2, double phi1, double phi2,
335:                             const std::vector<double> &kt, const std::vector<double> &phi,
336:                             const std::vector<double> &y, double m1, double m2) {
337:   const unsigned int Kf      = Nf - 2;  // Central system multiplicity
338:   static const M4Vec beamsum = lts.pbeam1 + lts.pbeam2;
339:
340:   if (lts.decaytree.size() != Kf) {
341:     std::string str =
342:         "MContinuum::BNBuildKin: Decaytree first level topology "
343:         "=s " +
344:         std::to_string(lts.decaytree.size()) + " (should be " + std::to_string(Kf) +
345:         " for this process)!";
346:     throw std::invalid_argument(str);
347:   }
348:
349:   // Forward protons px,py
350:   M4Vec p1(pt1 * std::cos(phi1), pt1 * std::sin(phi1), 0, 0);
351:   M4Vec p2(pt2 * std::cos(phi2), pt2 * std::sin(phi2), 0, 0);
352:
353:   // Auxialary "difference momentum" q0 = p0 - p1 ...
354:   pkt_.resize(Kf - 1);
355:   for (const auto &i : indices(pkt_)) {
356:     pkt_[i] = M4Vec(kt[i] * std::cos(phi[i]), kt[i] * std::sin(phi[i]), 0, 0);
357:   }
358:
359:   // Apply linear system to get p
360:   std::vector<M4Vec> p(Kf, M4Vec(0, 0, 0, 0));
361:   BLinearSystem(p, pkt_, p1, p2);
362:
363:   // Set pz and E for central final states
364:   M4Vec sumP(0, 0, 0, 0);
365:   for (const auto &i : indices(p)) {
366:     const double m  = lts.decaytree[i].m_offshell;  // Note offshell!
367:     const double mt = msqrt(pow2(m) + p[i].Pt2());
368:     p[i].SetPxEt(mt * std::sinh(y[i]), mt * std::cosh(y[i]));
369:     sumP += p[i];
370:   }
371:
372:   // Check crude energy overflow
373:   if (sumP.E() > lts.sqrt_s) { return false; }
374:
375:   double p1z = gra::kinematics::SolvePz(m1, m2, pt1, pt2, sumP.Pz(), sumP.E(), lts.s);
376:   double p2z = -(sumP.Pz() + p1z);  // by momentum conservation
377:
378:   // Enforce scattering direction +p -> +p, -p -> -p (VERY RARE POLYNOMIAL
379:   // BRANCH FLIP)
380:   if (p1z < 0 || p2z > 0) { return false; }
381:
382:   // pz and E of protons/N*
383:   p1.SetPxEt(p1z, msqrt(pow2(m1) + pow2(pt1) + pow2(p1z)));
384:   p2.SetPxEt(p2z, msqrt(pow2(m2) + pow2(pt2) + pow2(p2z)));
385:
386:   // ------------------------------------------------------------------
387:   // Now boost if asymmetric beams
388:   if (std::abs(beamsum.Pz()) > 1e-6) {
389:     constexpr int sign = 1;  // positive -> boost to the lab
390:     kinematics::LorentzBoost(beamsum, lts.sqrt_s, p1, sign);
391:     kinematics::LorentzBoost(beamsum, lts.sqrt_s, p2, sign);
392:     kinematics::LorentzBoost(beamsum, lts.sqrt_s, sumP, sign);
393:
394:     for (const auto &i : indices(p)) { kinematics::LorentzBoost(beamsum, lts.sqrt_s, p[i], sign); }
395:   }
396:   // ------------------------------------------------------------------
397:
398:   // First branch kinematics
399:   lts.pfinal[1] = p1;  // Forward systems
400:   lts.pfinal[2] = p2;
401:   lts.pfinal[0] = sumP;  // Central system
402:
403:   double              sumM   = 0;
404:   const unsigned int offset = 3;
405:   for (const auto &i : indices(p)) {
406:     lts.decaytree[i].p4    = p[i];
407:     lts.pfinal[i + offset] = p[i];
408:     sumM += p[i].M();
409:   }
410:
411:   // ------------------------------------------------------------------
412:   // Kinematic checks
413:
414:   // Check we are above mass threshold
415:   if (sumP.M() < sumM) { return false; }
```



```
416:
417:   // Total 4-momentum conservation
418:   if (!gra::math::CheckEMC(beamsum - (lts.pfinal[1] + lts.pfinal[2] + lts.pfinal[0]))) {
419:     return false;
420:   }
421:
422:   // ------------------------------------------------------------------
423:
424:   // Treat decaytree recursively
425:   for (const auto &i : indices(lts.decaytree)) {
426:     if (!ConstructDecayKinematics(lts.decaytree[i])) { return false; }
427:   }
428:
429:   // Forward system fragmentation
430:   if (vetocuts.active) {  // N.B. need to fragment already now!, if cuts active
431:     if (!CEPForwardFragment()) { return false; }
432:   }
433:
434:   return GetLorentzScalars(Nf);
435: }
436:
437: // Construct the linear system
438: void MContinuum::BLinearSystem(std::vector<M4Vec> &p, const std::vector<M4Vec> &q, const M4Vec &p1f,
439:                                const M4Vec &p2f) const {
440:   /*
441:
442:   w = p1f + p2f
443:
444:   --------------------------------
445:   KF = 4
446:
447:   2p0 = ( q0 - w - p2 - p3)
448:   2p1 = (-q0 - w - p2 - p3)
449:   2p2 = (-q1 - w - p0 - p3)
450:   2p3 = (-q2 - w - p0 - p1)
451:
452:   {2 0 1 1
453:    0 2 1 1
454:    1 0 2 1
455:    1 1 0 2}
456:
457:   --------------------------------
458:   KF = 3
459:
460:   2p0 = ( q0 - w - p2)
461:   2p1 = (-q0 - w - p2)
462:   2p2 = (-q1 - w - p0)
463:
464:   {2 0 1
465:    0 2 1
466:    1 0 2}
467:
468:   --------------------------------
469:   KF = 2
470:
471:   2p0 = ( q0 - w)
472:   2p1 = (-q0 - w)
473:
474:   {2 0
475:    0 2}
476:
477:   --------------------------------
478:   */
479:
480:   static const std::vector<std::vector<std::vector<double>>> A = {
481:       {{1.0 / 2.0, 0.0}, {0.0, 1.0 / 2.0}},
482:
483:       {{2.0 / 3.0, 0.0, -1.0 / 3.0},
484:        {1.0 / 6.0, 1.0 / 2.0, -1.0 / 3.0},
485:        {-1.0 / 3.0, 0.0, 2.0 / 3.0}},
486:
487:       {{7.0 / 8.0, 1.0 / 8.0, -1.0 / 2.0, -1.0 / 4.0},
488:        {3.0 / 8.0, 5.0 / 8.0, -1.0 / 2.0, -1.0 / 4.0},
489:        {-1.0 / 8.0, 1.0 / 8.0, 1.0 / 2.0, -1.0 / 4.0},
490:        {-5.0 / 8.0, -3.0 / 8.0, 1.0 / 2.0, 3.0 / 4.0}},
491:
492:       {{11.0 / 10.0, 3.0 / 10.0, -3.0 / 5.0, -2.0 / 5.0, -1.0 / 5.0},
493:        {3.0 / 5.0, 4.0 / 5.0, -3.0 / 5.0, -2.0 / 5.0, -1.0 / 5.0},
494:        {1.0 / 10.0, 3.0 / 10.0, 2.0 / 5.0, -2.0 / 5.0, -1.0 / 5.0},
495:        {-2.0 / 5.0, -1.0 / 5.0, 2.0 / 5.0, 3.0 / 5.0, -1.0 / 5.0},
496:        {-9.0 / 10.0, -7.0 / 10.0, 2.0 / 5.0, 3.0 / 5.0, 4.0 / 5.0}},
497:
498:       {{4.0 / 3.0, 1.0 / 2.0, -2.0 / 3.0, -1.0 / 2.0, -1.0 / 3.0, -1.0 / 6.0},
```



```
499:        {5.0 / 6.0, 1.0 / 1.0, -2.0 / 3.0, -1.0 / 2.0, -1.0 / 3.0, -1.0 / 6.0},
500:        {1.0 / 3.0, 1.0 / 2.0, 1.0 / 3.0, -1.0 / 2.0, -1.0 / 3.0, -1.0 / 6.0},
501:        {-1.0 / 6.0, 0.0, 1.0 / 3.0, 1.0 / 2.0, -1.0 / 3.0, -1.0 / 6.0},
502:        {-2.0 / 3.0, -1.0 / 2.0, 1.0 / 3.0, 1.0 / 2.0, 2.0 / 3.0, -1.0 / 6.0},
503:        {-7.0 / 6.0, -1.0 / 1.0, 1.0 / 3.0, 1.0 / 2.0, 2.0 / 3.0, 5.0 / 6.0}},
504:
505:       {{11.0 / 7.0, 5.0 / 7.0, -5.0 / 7.0, -4.0 / 7.0, -3.0 / 7.0, -2.0 / 7.0, -1.0 / 7.0},
506:        {15.0 / 14.0, 17.0 / 14.0, -5.0 / 7.0, -4.0 / 7.0, -3.0 / 7.0, -2.0 / 7.0, -1.0 / 7.0},
507:        {4.0 / 7.0, 5.0 / 7.0, 2.0 / 7.0, -4.0 / 7.0, -3.0 / 7.0, -2.0 / 7.0, -1.0 / 7.0},
508:        {1.0 / 14.0, 3.0 / 14.0, 2.0 / 7.0, 3.0 / 7.0, -3.0 / 7.0, -2.0 / 7.0, -1.0 / 7.0},
509:        {-3.0 / 7.0, -2.0 / 7.0, 2.0 / 7.0, 3.0 / 7.0, 4.0 / 7.0, -2.0 / 7.0, -1.0 / 7.0},
510:        {-13.0 / 14.0, -11.0 / 14.0, 2.0 / 7.0, 3.0 / 7.0, 4.0 / 7.0, 5.0 / 7.0, -1.0 / 7.0},
511:        {-10.0 / 7.0, -9.0 / 7.0, 2.0 / 7.0, 3.0 / 7.0, 4.0 / 7.0, 5.0 / 7.0, 6.0 / 7.0}},
512:
513:       {{29.0 / 16.0, 15.0 / 16.0, -3.0 / 4.0, -5.0 / 8.0, -1.0 / 2.0, -3.0 / 8.0, -1.0 / 4.0,
514:         -1.0 / 8.0},
515:        {21.0 / 16.0, 23.0 / 16.0, -3.0 / 4.0, -5.0 / 8.0, -1.0 / 2.0, -3.0 / 8.0, -1.0 / 4.0,
516:         -1.0 / 8.0},
517:        {13.0 / 16.0, 15.0 / 16.0, 1.0 / 4.0, -5.0 / 8.0, -1.0 / 2.0, -3.0 / 8.0, -1.0 / 4.0,
518:         -1.0 / 8.0},
519:        {5.0 / 16.0, 7.0 / 16.0, 1.0 / 4.0, 3.0 / 8.0, -1.0 / 2.0, -3.0 / 8.0, -1.0 / 4.0,
520:         -1.0 / 8.0},
521:        {-3.0 / 16.0, -1.0 / 16.0, 1.0 / 4.0, 3.0 / 8.0, 1.0 / 2.0, -3.0 / 8.0, -1.0 / 4.0,
522:         -1.0 / 8.0},
523:        {-11.0 / 16.0, -9.0 / 16.0, 1.0 / 4.0, 3.0 / 8.0, 1.0 / 2.0, 5.0 / 8.0, -1.0 / 4.0,
524:         -1.0 / 8.0},
525:        {-19.0 / 16.0, -17.0 / 16.0, 1.0 / 4.0, 3.0 / 8.0, 1.0 / 2.0, 5.0 / 8.0, 3.0 / 4.0,
526:         -1.0 / 8.0},
527:        {-27.0 / 16.0, -25.0 / 16.0, 1.0 / 4.0, 3.0 / 8.0, 1.0 / 2.0, 5.0 / 8.0, 3.0 / 4.0,
528:         7.0 / 8.0}}};
529:
530:   // Construct vector b
531:   const unsigned int Kf = p.size();  // Number of central system particles
532:   std::vector<M4Vec> b(Kf);
533:   const M4Vec        p1p2sum = p1f + p2f;
534:
535:   for (const auto &i : indices(b)) {
536:     if (i == 0) {
537:       b[i] = q[0] - p1p2sum;
538:     } else {
539:       b[i] = q[i - 1] * (-1.0) - p1p2sum;
540:     }
541:   }
542:
543:   // Apply linear system p = Ab to get px,py components for each p[i]
544:   const unsigned int index = Kf - 2;  // -2 because of C++
545:   for (const auto &i : indices(p)) {
546:     for (const auto &j : indices(b)) {
547:       p[i] += b[j] * A[index][i][j];  // notice plus
548:     }
549:   }
550: }
551:
552: // ------------------------------------------------------------------------
553: // MATLAB symbolic code to print out the system
554: // ------------------------------------------------------------------------
555: /*
556: % Number of particles in the central system
557: KFMAX = 8;
558:
559: fprintf('static const std::vector<std::vector<std::vector<double>>> A = \n{ ');
560: for KF = 2:KFMAX
561:
562: % Create system matrix A
563: A = ones(KF);
564: A(1:2,1:2) = [2 0; 0 2]; % Starting corner
565:
566: % Lower triangle
567: for i = 1:KF-2
568:     m = ones(1,KF);
569:     m(i+1) = 0;
570:     m(i+2) = 2;
571:     A(2+i,:) = m;
572: end
573:
574: % Invert
575: I = inv(sym(A));
576:
577: % Plot C++ style
578: fprintf('\n');
579: fprintf('{');
580: for i = 1:size(A,1)
581:         fprintf('{');
```



```
582:         for j = 1:size(A,2)
583:                 if (j < size(A,2))
584:                         mark = ', ';
585:                 else
586:                         mark = '';
587:                 end
588:                 [N,D] = numden(I(i,j));
589:                 if (N == 0)
590:                 fprintf('0.0%s', mark);
591:                 else
592:                 fprintf('%0.1f/%0.1f%s', N, D, mark);
593:                 end
594:         end
595:         if (i < size(A,2))
596:                 mark = '},';
597:         else
598:                 if (KF < KFMAX)
599:                         emark = ',';
600:                 else
601:                         emark = '';
602:                 end
603:                 mark = sprintf('}}%s', emark);
604:         end
605:         fprintf('%s\n', mark);
606: end
607:
608: end
609: fprintf('};\n');
610: */
611:
612: // (3*Nf-4)-Dim Integral Volume [phi1] x [phi2] x [phi_k1] x [phi_k2] x ... x
613: // [phi_{Kf-1}]
614: //                                [pt1] x [pt2] x [kt1] x [kt2] x ... x [kt{Kf-1}]
615: //                                xÂ [y3] x [y4] xÂ ... x [yN]
616: //
617: // For reference, Nf = 4 special case in:
618: // [REFERENCE: Lebiedowicz, Szczurek, arxiv.org/abs/0912.0190]
619: double MContinuum::BNIntegralVolume() const {
620:   // Number of central states
621:   const unsigned int Kf = lts.decaytree.size();
622:
623:   // Forward leg integration
624:   const double forward_volume = ForwardVolume();
625:
626:   // Intermediate kt volume
627:   double KT_vol = std::pow(gcuts.kt_max - gcuts.kt_min, Kf - 1);
628:
629:   // phi angle volume
630:   KT_vol *= std::pow(2.0 * PI, Kf - 1);
631:
632:   // rapidity volume
633:   KT_vol *= std::pow(gcuts.rap_max - gcuts.rap_min, Kf);
634:
635:   return KT_vol * forward_volume;
636: }
637:
638:
639: double MContinuum::BNPhaseSpaceWeight() const {
640:   // Jacobian, close to constant 0.5
641:   const double J = 1.0 / std::abs(lts.pfinal[1].Pz() / lts.pfinal[1].E() -
642:                                   lts.pfinal[2].Pz() / lts.pfinal[2].E());
643:
644:   // Number of final states
645:   const unsigned int Nf = lts.decaytree.size() + 2;
646:
647:   // Intermediate "difference"
648:   // kt factors from \prod_i d^2 k_i = \prod_i dphi_i kt_i dkt_i
649:   double PROD = 1.0;
650:   for (const auto &i : indices(pkt_)) { PROD *= pkt_[i].Pt(); }
651:
652:   const double factor = (1.0 / std::pow(2, 2 * (Nf - 2))) * (1.0 / std::pow(2.0 * PI, 3 * Nf - 4)) *
653:                         (lts.pfinal[1].Pt() / (2.0 * lts.pfinal[1].E())) *
654:                         (lts.pfinal[2].Pt() / (2.0 * lts.pfinal[2].E())) * J * PROD;
655:   return factor;
656: }
657:
658: }  // namespace gra
659:
```

```
// Abstract process class
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

// C++
#include <algorithm>
#include <cmath>
#include <complex>
#include <future>
#include <iostream>
#include <iterator>
#include <map>
#include <random>
#include <regex>
#include <stdexcept>
#include <vector>

// Own
#include "Granitti/MAux.h"
#include "Granitti/MFactorized.h"
#include "Granitti/MForm.h"
#include "Granitti/MFragment.h"
#include "Granitti/MGlobals.h"
#include "Granitti/MHl.h"
#include "Granitti/MH2.h"
#include "Granitti/MHELMatrix.h"
#include "Granitti/MKinematics.h"
#include "Granitti/MMath.h"
#include "Granitti/MMatrix.h"
#include "Granitti/MProcess.h"
#include "Granitti/MSpin.h"
#include "Granitti/MTimer.h"
#include "Granitti/MUserCuts.h"

// HepMC3
#include "HepMC3/FourVector.h"
#include "HepMC3/GenEvent.h"
#include "HepMC3/GenParticle.h"
#include "HepMC3/GenVertex.h"
#include "HepMC3/Print.h"

// Libraries
#include "json.hpp"
#include "rang.hpp"

using gra::aux::indices;
using gra::math::abs2;
using gra::math::msqrt;
using gra::math::pow2;
using gra::math::zi;

namespace gra {

void MProcess::PrintSetup() const {
  std::cout << std::endl;
  std::cout << rang::style::bold << "Process setup:" << rang::style::reset << std::endl
            << std::endl;
  std::cout << "- Random seed:      " << random.GetSeed() << std::endl;
  std::cout << "- Initial state:    " << lts.beam1.name << " " << lts.beam2.name << std::endl;
  printf("- Beam energies:   [%0.1f %0.1f] GeV \n", lts.pbeam1.E(), lts.pbeam2.E());
  printf("- CMS energy:       %0.1f GeV\n", lts.sqrt_s);
  std::cout << "- Process:          " << PROCESS << rang::fg::green << " [";

  for (const auto &str : ProcPtr.GetProcessDescriptor(PROCESS)) { std::cout << str << " | "; }
  std::cout << "]" << rang::fg::reset << std::endl << std::endl;

  // Subprocess
  std::cout << rang::style::bold << "Subprocess parameters:" << rang::style::reset << std::endl
            << std::endl;
  std::cout << "- Pomeron loop screening:  " << std::boolalpha << SCREENING << std::endl;

  // All other than inclusive processes
  if (ProcPtr.ISTATE != "X") {
    std::cout << "- Final state:             " << DECAYMODE << std::endl;
    std::cout << "- Proton N* excitation:    " << std::boolalpha << EXCITATION;

    if (EXCITATION == 0) {
      std::cout << rang::fg::green << "  <elastic>" << rang::fg::reset << std::endl;
    }
    if (EXCITATION == 1) {
      std::cout << rang::fg::green << "  <single>" << rang::fg::reset << std::endl;
    }
```



```
    if (EXCITATION == 2) {
      std::cout << rang::fg::green << "  <double>" << rang::fg::reset << std::endl;
    }
  }
  if (FLATAMP != 0) { std::cout << "- Flat amplitude mode:     " << FLATAMP << std::endl; }

  std::cout << std::endl << std::endl;
}


// Cascaded phase-space factor [VOLUME] x [MC INTEGRAL] / (2PI)
double MProcess::CascadePS() const {
  // Cascade resonances phase-space
  double product    = 1.0;
  double product2pi = 1.0;
  double volume     = 1.0;
  int    N_final    = 0;
  for (const auto &i : indices(lts.decaytree)) {
    CalculatePhaseSpace(lts.decaytree[i], product, product2pi, volume, N_final);
  }
  return volume * product / product2pi;
}

// Recursive function to calculate phasespace weights
void MProcess::CalculatePhaseSpace(const gra::MDecayBranch &branch, double &product,
                                   double &product2pi, double &volume, int &N_final) const {
  // There is decay information and this is activated by the amplitude
  if (branch.PS_active == true && branch.W.Integral() > 0) {
    product *= branch.W.Integral();
    product2pi *= (2 * math::PI);

    // Flat mass^2 interval
    double M_sum = 0;
    for (const auto &i : indices(branch.legs)) {
      M_sum += branch.legs[i].p.mass;
    }  // Daughters give lower limit

    const double MIN = std::max(pow2(M_sum), pow2(branch.p.mass - branch.p.width * OFFSHELL));
    const double MAX = pow2(branch.p.mass + branch.p.width * OFFSHELL);
    volume *= (MAX - MIN);

    for (const auto &i : indices(branch.legs)) {
      CalculatePhaseSpace(branch.legs[i], product, product2pi, volume, N_final);
    }
  }
  if (branch.legs.size() == 0) { ++N_final; }  // Final state particle
}

// Amplitude squared
double MProcess::GetAmp2() {
  double amp2 = (FLATAMP == 0) ? S3ScreenedAmp2() : GetFlatAmp2(lts);

  // ------------------------------------------------
  // ** Custom sampling control initiated by amplitudes **
  if (lts.FORCE_FLATMASS2 && !FLATMASS2_user) { FLATMASS2 = true; }
  if (lts.FORCE_OFFSHELL >= 0 && !OFFSHELL_user) { OFFSHELL = lts.FORCE_OFFSHELL; }
  // ------------------------------------------------

  return amp2;
}

// Pomeron Loop Screened Amplitude
//
// ======xxxxxxxxx======>
//       *       *
//       *     -p1t + kt = qt1
//       *       *
//       * kt          x==>
//       *       *
//       *     -p2t - kt = qt2
//       *       *
// ======xxxxxxxxx======>
double MProcess::S3ScreenedAmp2() {
  // Eikonal Loop Screening not on
  if (SCREENING == false) { return ProcPtr.GetBareAmplitude2(lts); }

  // Elastic scattering, return directly eikonalized amplitude squared itself
  if (ProcPtr.CHANNEL == "EL") { return abs2(Eikonal.MSA.Interpolate1D(-lts.t)); }

  // ------------------------------------------------------------------
  // First evaluate bare amplitudes to lts.hamp
  ProcPtr.GetBareAmplitude2(lts);
```



```

  if (lts.hamp.size() == 0) {
    const std::string str =
      "MProcess::S3ScreenedAmp2: Amplitude does not support screening, lts.hamp.size == 0";
    throw std::invalid_argument(str);
  }

  // Save born amplitudes
  const std::vector<std::complex<double>> hamp_0 = lts.hamp;
  // ------------------------------------------------------------------

  // Proton pt vectors
  const std::vector<double> p1T = (lts.pfinal[1].Px(), lts.pfinal[1].Py());
  const std::vector<double> p2T = (lts.pfinal[2].Px(), lts.pfinal[2].Py());

  // Save old kinematics
  lts.pfinal_orig = lts.pfinal;

  // @@ C++11 handles multithreaded static initialization @@
  // Discretization steps
  const static double StepKT =
      (Eikonal.Numerics.MaxLoopKT - Eikonal.Numerics.MinLoopKT) / Eikonal.Numerics.NumberLoopKT;
  const double       MinPhi  = 0.0;
  const double       MaxPhi  = 2.0 * gra::math::PI;
  const static double StepPhi = (MaxPhi - MinPhi) / Eikonal.Numerics.NumberLoopPHI;

  // Init 2D-Simpson weight matrix (will be calculated only once, being
  // static)
  const static MMatrix<double> WSimpson =
      gra::math::Simpson3DWeight2D(Eikonal.Numerics.NumberLoopPHI, Eikonal.Numerics.NumberLoopKT);

  // NOTE N + 1, init with zero!
  std::vector<MMatrix<std::complex<double>>> f_hamp(
      lts.hamp.size(), MMatrix<std::complex<double>>(Eikonal.Numerics.NumberLoopPHI + 1,
                                                     Eikonal.Numerics.NumberLoopKT + 1, 0.0));

  // 2D-integral
  //
  // \int d^2kt A_eik(kt^2) A(kt1,kt2)
  // = \int d\phi \int dkt * kt * A_el(kt^2) * A(kt1,kt2)
  //
  //          *2
  //     *3   |    *1
  //          |phi
  //  *4----------*0/8  ---> kt --->
  //          |
  //     *5   |    *7
  //          *6
  //

  for (std::size_t i = 0; i < Eikonal.Numerics.NumberLoopPHI + 1; ++i) {
    const double phi = MinPhi + i * StepPhi;

    for (std::size_t j = 0; j < Eikonal.Numerics.NumberLoopKT + 1; ++j) {
      const double kt  = Eikonal.Numerics.MinLoopKT + j * StepKT;
      const double kt2 = gra::math::pow2(kt);

      // ------------------------------------------------------------------

      // Get screening amplitude
      // (compiler will optimize this outside the loop)
      const std::complex<double> A_eik = Eikonal.MSA.Interpolate1D(kt2);

      // 0. Construct loop 2D kT-vector
      const std::vector<double> kt_{kt * std::cos(phi), kt * std::sin(phi)};

      // 1. New proton pt vectors
      const std::vector<double> p1p = {p1T[0] - kt_[0], p1T[1] - kt_[1]};
      const std::vector<double> p2p = {p2T[0] + kt_[0], p2T[1] + kt_[1]};

      // 2. Update kinematics
      if (!LoopKinematics(p1p, p2p)) {
        continue;  // not valid kinematically
      }

      // 3. Get new amplitudes to lts.hamp
      ProcPtr.GetBareAmplitude2(lts);
      // ------------------------------------------------------------------

      // Loop over all helicity amplitudes
      for (const auto &h : indices(lts.hamp)) {
        f_hamp[h][i][j] = A_eik * lts.hamp[h] * kt;  // note x kt (jacobian)
      }
```



```
    }  // inner-loop
  }    // outer-loop

  // Normalization
  const std::complex<double> norm = zi / (8.0 * gra::math::PIPI * lts.s);

  // Loop over amplitudes
  std::vector<std::complex<double>> hamp_loop(lts.hamp.size(), 0.0);

  for (const auto &h : indices(lts.hamp)) {
    hamp_loop[h] = norm * gra::math::Simpson3DIntegral2D(f_hamp[h], WSimpson, StepPhi, StepKT);
  }

  // Update back to tree level kinematics
  LoopKinematics(p1T, p2T);

  // ------------------------------------------------------------
  // Final amplitude (squared)

  // Not Durham-QCD
  if (ProcPtr.ISTATE != "gg") {
    // Separate (incoherent) sum
    double amp2 = 0.0;
    for (const auto &h : indices(lts.hamp)) { amp2 += abs2(hamp_0[h] + hamp_loop[h]); }

    // Initial state helicity average, if we have all helicity amplitudes
    if (lts.hamp.size() != 1) { amp2 /= 4; }
    return amp2;

    // Durham-QCD
  } else {
    // Coherent sum
    std::complex<double> A = 0.0;
    for (const auto &h : indices(lts.hamp)) { A += hamp_0[h] + hamp_loop[h]; }
    // Helicity average already taken care of

    return abs2(A);
  }
}

// Set CMS energy and beam particle 4-vectors
void MProcess::SetInitialState(const std::vector<std::string> &beam,
                               const std::vector<double> &     energy) {
  if (beam.size() != 2) {
    throw std::invalid_argument("MProcess::SetInitialState: Input BEAM vector not dim 2!");
  }
  if (energy.size() != 2) {
    throw std::invalid_argument("MProcess::SetInitialState: Input ENERGY vector not dim 2!");
  }

  lts.beam1 = lts.PDG.FindByPDGName(beam[0]);
  lts.beam2 = lts.PDG.FindByPDGName(beam[1]);

  // Beam particle 4-momenta re-setup with safety threshold if fixed target
  // setup
  const double E1 = std::max(energy[0], 1.001 * lts.beam1.mass);
  const double E2 = std::max(energy[1], 1.001 * lts.beam2.mass);

  SetBeamEnergies(E1, E2);
}

// Beam needs to be set before this!
void MProcess::SetBeamEnergies(double E1, double E2) {
  if (lts.beam1.pdg == 0 || lts.beam2.pdg == 0) {
    throw std::invalid_argument("MProcess::SetBeamEnergies: Beam PDG particles not set yet!");
  }

  // Beam 4-momentum (px,py,pz,E)
  lts.pbeam1 = M4Vec(0, 0, msqrt(pow2(E1) - pow2(lts.beam1.mass)), E1);   // positive z-axis
  lts.pbeam2 = M4Vec(0, 0, -msqrt(pow2(E2) - pow2(lts.beam2.mass)), E2);  // negative z-axis

  // Mandelstam s
  lts.s      = (lts.pbeam1 + lts.pbeam2).M2();
  lts.sqrt_s = msqrt(lts.s);

  if (lts.sqrt_s < (lts.beam1.mass + lts.beam2.mass)) {
    std::string str = "MProcess::SetBeamEnergies: Error with input CMS energy: " +
                      std::to_string(lts.sqrt_s) + " GeV < initial state masses!";
    throw std::invalid_argument(str);
  }

  printf("MProcess::SetBeamEnergies: beam: [%s, %s], energy = [%0.1f, %0.1f] \n",
```

```
333:          lts.beam1.name.c_str(), lts.beam2.name.c_str(), E1, E2);
334: }
335:
336:
337: // Return "flat/special" matrix element squared |A|^2
338: // for evaluating the phase space and other special purposes
339: double MProcess::GetFlatAmp2(const gra::LORENTZSCALAR &lts) const {
340:   double W = 1.0;
341:
342:   // Ansatz: |A|^2 ~ exp(bt1) exp(bt2)
343:   if (FLATAMP == 1) {
344:     W = pow2(lts.s) * std::exp(PARAM_FLAT::b * lts.t1) * std::exp(PARAM_FLAT::b * lts.t2);
345:   }
346:   // Ansatz: |A|^2 ~ exp(bt1) exp(bt2) / sqrt(shat)
347:   else if (FLATAMP == 2) {
348:     W = pow2(lts.s) * std::exp(PARAM_FLAT::b * lts.t1) * std::exp(PARAM_FLAT::b * lts.t2) /
349:         msqrt(lts.s_hat);
350:   }
351:   // Ansatz: |A|^2 ~ exp(bt1) exp(bt2) / shat
352:   else if (FLATAMP == 3) {
353:     W = pow2(lts.s) * std::exp(PARAM_FLAT::b * lts.t1) * std::exp(PARAM_FLAT::b * lts.t2) /
354:         lts.s_hat;
355:   }
356:   // Constant
357:   else if (FLATAMP == 4) {
358:     W = 1.0;
359:   } else {
360:     // Throw an error, unknown mode
361:     std::string str =
362:         "MProcess::GetFlatAmp2: Unknown FLATAMP ({|A|^2 is 0 = off, 1 = "
363:         "exp(b(t1+t2)), 2 = exp(b(t1+t2))/shat^(1/2), 3 = exp(b(t1+t2))/shat, 4 = 1.0) "
364:         ": input was " +
365:         std::to_string(FLATAMP);
366:     throw std::invalid_argument(str);
367:   }
368:   return W;
369: }
370:
371: // Set decaymode
372: void MProcess::SetDecayMode(std::string str) {
373:   // Clear decaytree
374:   lts.decaytree.clear();
375:
376:   // ------------------------------------------------------------------
377:   // 0. Check if contains only spaces (or is empty)
378:   std::string           check   = str;
379:   std::string::iterator end_pos = std::remove(check.begin(), check.end(), ' ');
380:   check.erase(end_pos, check.end());
381:
382:   // Can be empty only for <Q>~class (2->2/inclusive) processes or <F>
383:   // factorized phase space
384:   if (check.size() == 0 && !(PROCESS.find("<Q>") != std::string::npos) &&
385:       !(PROCESS.find("<F>") != std::string::npos)) {
386:     throw std::invalid_argument(
387:         "MProcess::SetDecayMode: DECAY string input is "
388:         "empty (can be only for <Q> or "
389:         "<F> class)!");
390:   }
391:
392:   // ------------------------------------------------------------------
393:   // 1. Syntax check, we must have same amount of left ( and right ) brackets
394:   std::vector<std::string> words = gra::aux::Extract(str);
395:
396:   unsigned int arrows = 0;
397:   unsigned int L      = 0;
398:   unsigned int R      = 0;
399:   for (std::string::size_type i = 0; i < str.size(); ++i) {
400:     if (str[i] == '(') ++L;
401:     if (str[i] == ')') ++R;
402:     if (str[i] == '>') ++arrows;
403:   }
404:   if (L != R) {
405:     std::string strerr = "MProces::SetDecayMode: ERROR: Decay tree has " + std::to_string(L) +
406:                          " left brackets ( and " + std::to_string(R) + " right brackets ) !";
407:     throw std::invalid_argument(strerr);
408:   }
409:   if (arrows != L) {
410:     std::string strerr =
411:         "MProces::SetDecayMode: ERROR: Decay tree syntax with "
412:         "arrows > not equal to ( "
413:         ") brackets!";
414:     throw std::invalid_argument(strerr);
415:   }
```



```
416:
417:   // ============================================================
418:   // ** Read decay by recursion **
419:   lts.PDG.TokenizeProcess(str, 0, lts.decaytree);
420:   // ============================================================
421:
422:   // Save decaymode string
423:   DECAYMODE = str;
424:
425:   // Calculate symmetry factor
426:   CalculateSymmetryFactor();
427:
428:   // Remind the user
429:   if (lts.decaytree.size() > 2) {
430:     gra::aux::PrintWarning();
431:     std::cout << "Reminder: Resonance decay |matrix element 1->K|^2 is "
432:                  "non-factorizable from the phase space for K = " +
433:                  std::to_string(lts.decaytree.size()) + " > 2!"
434:               << std::endl;
435:     std::cout << "Use &> arrow for fully separating 2->3 [+] 1->K" << std::endl;
436:     std::cout << std::endl;
437:   }
438: }
439:
440: // Count QFT statistical (combinatorial) factor for identical final states
441: //
442: // BASED here counting the first level final states
443: // Example: yyyy             -> 4!
444: //          pi+pi-pi+pi-     -> 2!2! (CHECK THIS!)
445: //          pi+pi-pi+pi-pi+- -> 3!3!1!
446: //          yy               -> 2!
447: //          pi+pi-K+K-       -> 1
448: //
449: void MProcess::CalculateSymmetryFactor() {
450:   S_factor = 1.0;  // Init with 1
451:
452:   std::vector<bool> marked(lts.decaytree.size(), false);
453:
454:   // Collect here multiplicities
455:   std::vector<int> multiplicities;
456:
457:   for (const auto &i : indices(lts.decaytree)) {
458:     int n   = 1;
459:     marked[i] = true;
460:     for (const auto &j : indices(lts.decaytree)) {
461:       if (i != j && lts.decaytree[i].p.pdg == lts.decaytree[j].p.pdg && marked[j] == false) {
462:         marked[j] = true;
463:         ++n;
464:       }
465:     }
466:     S_factor *= gra::math::factorial(n);
467:   }
468: }
469:
470: // Read resonance branching fractions
471: void MProcess::SetupBranching() {
472:   aux::PrintBar('.');
473:
474:   if (lts.RESONANCES.size() == 0) {  // Empty one is not treated
475:     std::cout << rang::fg::red << "MProcess::SetupBranching: lts.RESONANCES.size() == 0 !"
476:               << rang::fg::reset << std::endl;
477:     return;
478:   }
479:   if (lts.decaytree.size() == 0) {
480:     std::cout << rang::fg::red << "MProcess::SetupBranching: decaytree.size() == 0 !"
481:               << rang::fg::reset << std::endl;
482:     return;
483:   }
484:
485:   // Loop over resonances
486:   for (auto const &xpoint : lts.RESONANCES) {
487:     // Take the resonance
488:     gra::PARAM_RES res = xpoint.second;
489:
490:     // Process helicity decay information
491:     std::vector<MParticle> daughter;
492:     for (const auto &i : indices(lts.decaytree)) { daughter.push_back(lts.decaytree[i].p); }
493:     res.hel = ProcessHelicityDecay(res.p, daughter);
494:
495:     // Set the updated resonance
496:     lts.RESONANCES[xpoint.first] = res;
497:   }
498:
```



```
499:   // Recursively over the decaytree
500:   for (const auto &i : indices(lts.decaytree)) {
501:     if (lts.decaytree[i].legs.size() != 0) {  // Has any daughters?
502:       ProcessHelicityTree(lts.decaytree[i]);
503:     }
504:   }
505: }
506:
507: // Recursive function
508: //
509: void MProcess::ProcessHelicityTree(MDecayBranch &branch) {
510:   // Process decay information
511:   std::vector<MParticle> daughter;
512:   for (const auto &i : indices(branch.legs)) { daughter.push_back(branch.legs[i].p); }
513:   branch.hel = ProcessHelicityDecay(branch.p, daughter);
514:
515:   // Recursion
516:   for (const auto &i : indices(branch.legs)) {
517:     if (branch.legs[i].legs.size() != 0) {  // Has any daughters?
518:       ProcessHelicityTree(branch.legs[i]);
519:     }
520:   }
521: }
522:
523: // Process decay helicity amplitude by
524: // reading alpha_ls and (tensor pomeron model) couplings from BRANCHING.json
525: //
526: HELMatrix MProcess::ProcessHelicityDecay(const MParticle &                 p,
527:                                          const std::vector<MParticle> &daughter) const {
528:   HELMatrix hc;
529:
530:   // Get decay daughter PDG ids
531:   std::vector<int> refdecay;
532:   for (const auto &i : indices(daughter)) { refdecay.push_back(daughter[i].pdg); }
533:
534:   // Setup branching fractions for resonances
535:   using json = nlohmann::json;
536:
537:   const std::string inputfile =
538:       gra::aux::GetBasePath(2) + "/modeldata/" + gra::MODELPARAM + "/BRANCHING.json";
539:   const std::string data = gra::aux::GetInputData(inputfile);
540:   json              j;
541:   try {
542:     j = json::parse(data);
543:   } catch (...) {
544:     std::string str = "MProcess::SetupBranching: Error parsing " + inputfile +
545:                       " (Check for extra/missing commas)";
546:     throw std::invalid_argument(str);
547:   }
548:
549:   // ------------------------------------------------------------------
550:
551:   const int pdg = p.pdg;
552:
553:   // Find out if we have this resonance in tables
554:   std::string PDG_STR = std::to_string(pdg);
555:
556:   // Did we find that resonance written down it in the Branching Tables
557:   bool found = j.count(PDG_STR);
558:
559:   if (found == true) {
560:     // Do we find it from PDG database
561:     std::string resonance_name = "not-found-from-PDG";
562:     try {
563:       MParticle part = lts.PDG.FindByPDG(pdg);
564:       resonance_name = part.name;
565:     } catch (...) {
566:       // continue;
567:     }
568:     std::cout << "MProcess::ProcessHelicityDecay: Resonance PDG = " + std::to_string(pdg) + " / " +
569:                  resonance_name
570:               << std::endl;
571:     std::cout << "Decay to ";
572:     for (const auto &i : indices(daughter)) { std::cout << daughter[i].pdg << " "; }
573:     std::cout << std::endl;
574:
575:     // Try to find the decaymode
576:     bool          found_decay_mode = false;
577:     const unsigned int MAXDECAYS   = 500;
578:     for (std::size_t i = 0; i < MAXDECAYS; ++i) {
579:       const std::string SID = std::to_string(i);
580:       std::vector<int>  decay;
581:       try {
```



```
582:         std::vector<int> vec = j[PDG_STR][SID]["PDG"];
583:         decay                = vec;
584:       } catch (...) {
585:         continue;  // Did not found anything with this
586:                    // index [*i*]
587:       }
588:
589:       if (std::is_permutation(refdecay.begin(), refdecay.end(), decay.begin())) {
590:         // Found matching decay
591:         hc.BR          = j[PDG_STR][SID]["BR"];
592:         hc.P_conservation = j[PDG_STR][SID]["P_conservation"];
593:         found_decay_mode  = true;
594:
595:         try {
596:           std::vector<double> temp = j[PDG_STR][SID]["g_decay_tensor"];
597:           hc.g_decay_tensor        = temp;
598:         } catch (...) {
599:           // Did not found uset set Tensor Pomeron decay coupling array, do nothing
600:         }
601:
602:         // ------------------------------------------------------------------
603:         // Construct helicity ls-coupling matrix (put default 1.0)
604:         // indexed directly by ls values
605:         MMatrix<std::complex<double>> alpha(20, 20, 1.0);
606:         MMatrix<bool>                 alpha_set(20, 20, false);
607:
608:         // Allow 50 coupling parameter (1,s) pairs (more than
609:         // enough)
610:         for (std::size_t a = 0; a < 50; ++a) {
611:           try {
612:             const unsigned int l  = j[PDG_STR][SID]["alpha_ls"][a][0];
613:             const unsigned int s  = j[PDG_STR][SID]["alpha_ls"][a][1];
614:             const double       Re = j[PDG_STR][SID]["alpha_ls"][a][2];
615:             const double       Im = j[PDG_STR][SID]["alpha_ls"][a][3];
616:
617:             // Set value
618:             alpha[l][s] = Re + zi * Im;
619:
620:             // Mark it set
621:             alpha_set[l][s] = true;
622:
623:             std::cout << "Found ls-coupling from BRANCHING.json: "
624:                       << "[l=" << l << ",s=" << s << "] = " << alpha[l][s] << " (Re,Im)"
625:                       << std::endl;
626:           } catch (...) { continue; }
627:         }
628:         hc.alpha     = alpha;
629:         hc.alpha_set = alpha_set;
630:         break;
631:       }
632:     }
633:
634:     if (!found_decay_mode) {
635:       gra::aux::PrintWarning();
636:       std::cout << rang::fg::red
637:                 << "WARNING: BRANCHING.json contains no information on this decay: " +
638:                        std::to_string(pdg) + " -> ";
639:
640:       for (const auto &i : indices(daughter)) { std::cout << daughter[i].pdg << " "; }
641:       std::cout << std::endl;
642:
643:       std::cout << "(setting up BR = 1.0, alpha_ls == 1.0, P_conservation = true)"
644:                 << rang::fg::reset << std::endl;
645:       hc.BR          = 1.0;
646:       hc.P_conservation = true;
647:       hc.alpha       = MMatrix<std::complex<double>>(20, 20, 1.0);
648:       hc.alpha_set   = MMatrix<bool>(20, 20, true);
649:     }
650:
651:     // ------------------------------------------------------------------
652:     // Init T-matrix
653:     if (daughter.size() == 2) {
654:       try {
655:         gra::spin::InitTMatrix(hc, p, daughter[0], daughter[1]);
656:       } catch (std::invalid_argument &e) {
657:         throw std::invalid_argument("Problem with resonance PDG = " + std::to_string(pdg) + " : " +
658:                                     e.what());
659:       }
660:     }
661:     // ------------------------------------------------------------------
662:
663:     // ------------------------------------------------------------------
664:     // Calculate equivalent decay coupling for 2-body cases
```

```
665:     //
666:     // In 2-body case, the decay phase space and decay amplitude squared
667:     // factorize
668:
669:     bool TP_computed = false;
670:     if (daughter.size() == 2) {
671:       // Calculate phase space part:
672:       // Gamma = PS * |A_decay|^2 then with |A_decay|^2 = 1 <=> Gamma = PS
673:
674:       const double amp2 = 1.0;
675:       const double sym  = 1.0;
676:
677:       double PS = gra::kinematics::PDW2body(pow2(p.mass), pow2(daughter[0].mass),
678:                                             pow2(daughter[1].mass), amp2, sym);
679:
680:       // ------------------------------------------------------------
681:       // Tensor Pomeron Model decay couplings for 2-body decays directly calculable from width and
682:       // BR
683:       if (hc.g_decay_tensor.size() == 0) {  // Not set yet
684:
685:         hc.g_decay_tensor = {
686:             MTensorPomeron::GDecay(p.spinX2 / 2, p.mass, p.width, daughter[0].mass, hc.BR)};
687:
688:         if (std::abs(PS) < ZERO_EPS) {
689:           // Try again with higher mother mass as a crude approximation, we might
690:           // be trying purely off-shell decay (such as f0(980) -> K+K-)
691:           const unsigned int N_width = 3;
692:           for (std::size_t i = 1; i <= N_width; ++i) {
693:             PS = gra::kinematics::PDW2body(pow2(p.mass + i * p.width), pow2(daughter[0].mass),
694:                                            pow2(daughter[1].mass), amp2, sym);
695:
696:             // ------------------------------------------------------------
697:             // Tensor Pomeron Model decay couplings
698:             hc.g_decay_tensor = {MTensorPomeron::GDecay(p.spinX2 / 2, p.mass + i * p.width, p.width,
699:                                                         daughter[0].mass, hc.BR)};
700:
701:             if (PS > ZERO_EPS) { break; }
702:           }
703:         }
704:         TP_computed = true;
705:       }
706:
707:       // BR \equiv Gamma/Gamma_tot = (PS * |A_decay|^2) / Gamma_tot
708:       // <=>
709:       // |A|^2 = BR * Gamma_tot / Gamma_PS and sqrt to get 'amplitude
710:       // level'
711:       //
712:       // ** GeV unit can be obtained by inspecting PDW2body function **
713:       hc.g_decay = msqrt(hc.BR * p.width / PS);
714:
715:       if (std::isnan(hc.g_decay) || std::isinf(hc.g_decay)) {
716:         std::string str =
717:             "MProcess::ProcessHelicityDecay: Kinematic coupling problem "
718:             "to '" +
719:             DECAYMODE + "' with resonance PDG " + p.name + "(" + std::to_string(p.pdg) + ")" +
720:             " (daughters too heavy?) (check RESONANCE, DECAYMODE "
721:             "and BRANCHING tables)";
722:         throw std::invalid_argument(str);
723:       }
724:       // ------------------------------------------------------------
725:
726:       // 3,4,... body cases {NOT TREATED YET, phase space and matrix
727:       // element do not factorize there}
728:     } else {
729:       hc.g_decay = 1.0;  // For the rest, put 1.0
730:     }
731:
732:     printf("(Mass, Full width):                  (%0.3E, %0.3E GeV) \n", p.mass, p.width);
733:     printf("Branching ratio || Partial width:   %0.3E || %0.3E GeV \n", hc.BR, hc.BR * p.width);
734:     printf("=> Computed decay coupling:          %0.3E \n", hc.g_decay);
735:     printf("=> Tensor Pomeron decay couplings:       [ ");
736:     for (const auto &i : indices(hc.g_decay_tensor)) { printf("%0.3E ", hc.g_decay_tensor[i]); }
737:     printf("] %s \n", TP_computed ? "(computed from J, width and BR)" : "");
738:
739:   } else {
740:     std::string str =
741:         "MProcess::ProcessHelicityDecay: Did not find any branching ratio data for resonance with "
742:         "PDG: " +
743:         std::to_string(p.pdg);
744:     throw std::invalid_argument(str);
745:   }
746:   aux::PrintBar(".");
747:
```



```
748:   return hc;
749: }
750:
751:
752: // Get intermediate off-shell mass
753: void MProcess::GetOffShellMass(const gra::MDecayBranch &branch, double &mass) {
754:   // If zero-width particle (electron, gamma, proton)
755:   if (branch.p.width < 1e-40) {
756:     mass = branch.p.mass;
757:     return;
758:   }
759:
760:   const unsigned int OUTERMAXTRIAL = 1e4;
761:   const unsigned int INNERMAXTRIAL = 1e4;
762:
763:   unsigned int outertrials = 0;
764:
765:   while (true) {
766:     // We have decay daughters
767:     double daughter_masses = 0;
768:     if (branch.legs.size() != 0) {
769:       // Find random daughter offshell masses from BW
770:       for (const auto &i : indices(branch.legs)) {
771:         daughter_masses += std::max(
772:             0.0,
773:             random.RelativisticBWRandom(branch.legs[i].p.mass, branch.legs[i].p.width, OFFSHELL));
774:       }
775:       const double safe_margin = 1e-5;  // GeV
776:       daughter_masses += safe_margin;
777:     }
778:
779:     // Pick mother offshell mass
780:     unsigned int innertrials = 0;
781:     while (true) {
782:       const double M = branch.p.mass;
783:       const double W = branch.p.width;
784:
785:       if (!FLATMASS2) {
786:         mass =
787:             std::max(daughter_masses, random.RelativisticBWRandom(M, W, OFFSHELL, daughter_masses));
788:       } else {
789:         mass = msqrt(random.U(std::max(pow2(daughter_masses), pow2(M - OFFSHELL * W)),
790:                               std::min(lts.s, pow2(M + OFFSHELL * W))));
791:       }
792:
793:       ++innertrials;
794:       if (mass >= daughter_masses) {
795:         return;  // all done
796:       }
797:       if (innertrials > INNERMAXTRIAL) {
798:         break;  // try again with different daughter masses
799:       }
800:     }
801:     ++outertrials;
802:
803:     // Impossible
804:     if (outertrials > OUTERMAXTRIAL) {
805:       std::string str;
806:       for (const auto &i : indices(branch.legs)) { str += branch.legs[i].p.name + " "; }
807:       str = "MProcess::GetOffShellMass: Kinematically impossible decay: " + branch.p.name + " > " +
808:             str;
809:       throw std::invalid_argument(str);
810:     }
811:   }
812: }
813:
814: // ------------------------------------------------------------------------
815:
816: // Proton continuum excitation
817: //
818: // Return true if OK
819: // Return false if FAILS
820: //
821: bool MProcess::ExciteContinuum(const M4Vec &nstar, gra::MDecayBranch &forward, double Q2_scale,
822:                                int B, int Q, const std::string &pt_distribution) {
823:   // Sanity check (2 x pion mass)
824:   if (msqrt(Q2_scale) < 0.3) {
825:     // return false; // not valid
826:   }
827:   const double QProb      = 2.0 / 3.0;  // Probability for a charged particle (isospin)
828:   const int    OUTERTRIAL = 100;
829:
830:   // Average multiplicity
```



```
831:   const double a0          = 0.0;
832:   double       b0          = 2.0;
833:   double       avgN        = a0 + b0 * (1 / QProb) * std::log(Q2_scale);
834:   int          outertrials = 0;
835:
836:   const double M = nstar.M();
837:
838:   while (true) {  // OUTER
839:
840:     // Draw fluctuating number of particles
841:     int N = 0;
842:     while (true) {
843:       N = random.PoissonRandom(avgN);
844:       // if (M < 10) { // low-mass smearing treatment
845:       for (std::size_t r = 0; r < 3; ++r) {
846:         //    N = PoissonRND(N);
847:       }
848:       //}
849:       if (nstar.M() > N * 0.14) break;  // Minimal mass threshold
850:     }
851:
852:     // Boundary conditions
853:     N = (N < 1) ? 1 : N;                      // At least 1
854:     N = (N >= std::abs(B)) ? N : std::abs(B);  // Baryon number
855:     N = (N >= std::abs(Q)) ? N : std::abs(Q);  // Charge conservation
856:
857:     // ------------------------------------------------------------------
858:     // Pick particles
859:
860:     std::vector<double> mass;
861:     std::vector<int>    pdgcode;
862:     if (!MFragment::PickParticles(M, N, B, 0, Q, mass, pdgcode, lts.PDG, random)) {
863:       ++outertrials;
864:       continue;
865:     }
866:
867:     // ------------------------------------------------------------------
868:     // Now decay >>
869:
870:     std::vector<M4Vec> p4;
871:
872:     double q = std::pow(N, 0.11 / 3);  // Powerlaw high-pt slope
873:     double T = 0.065;                  // Temperature
874:     if (etree.size() > 0) {  // For ND soft-cuts, "Temperature" rises ~ 1/impact parameter squared
875:       T *= std::pow(1 / (bt * bt), 0.10);
876:     }
877:     const double maxpt  = 15.0;  // Maximum Pt per particle
878:     const double lambda = 2.0;   // GeV^(-2), for exponential pt^2
879:     const double W =
880:         MFragment::TubeFragment(nstar, M, mass, p4, q, T, lambda, maxpt, random, pt_distribution);
881:
882:     if (W <= 0) {
883:       ++outertrials;
884:       if (outertrials > OUTERTRIAL) {
885:         // std::cout << "MProcess::ExciteContinuum: Outer loop failure!" <<
886:         // std::endl;
887:         return false;  // too many failures
888:       } else {
889:         continue;  // try again!
890:       }
891:     } else {
892:       // Re-check kinematics
893:       bool fail = false;
894:       for (const auto &i : indices(p4)) {
895:         if (std::isnan(p4[i].Rap())) {
896:           // std::cout << rang::fg::yellow <<
897:           // "MProcess::ExciteContinuum: Kinematics failure!" <<
898:           // rang::style::reset << std::endl;
899:           ++outertrials;
900:           if (outertrials > OUTERTRIAL) {
901:             // std::cout << "MProcess::ExciteContinuum: Outer
902:             // loop failure!" << std::endl;
903:             return false;  // too many failures
904:           }
905:           fail = true;
906:           break;
907:         }
908:       }
909:       if (fail) { continue; }
910:
911:       // Get corresponding PDG particles
912:       std::vector<gra::MParticle> p(p4.size());
913:       for (const auto &i : indices(p)) { p[i] = lts.PDG.FindByPDG(pdgcode[i]); }
```



```
914:
915:       // Create decaytree
916:       BranchForwardSystem(p4, p, nstar, forward);
917:       break;
918:     }
919:   }
920:   return true;
921: }
922:
923: // Proton low mass (N* like) 2-body and 3-body excitation
924: //
925: //
926: bool MProcess::ExciteNstar(const M4Vec &nstar, gra::MDecayBranch &forward, const MParticle &pbeam) {
927:   // Find random decaymode
928:   std::vector<int> pdgcodes;
929:   MFragment::NstarDecayTable(pbeam.chargeX3 / 3, nstar.M(), pdgcodes, random);
930:
931:   // Get corresponding PDG particles
932:   std::vector<gra::MParticle> p(pdgcodes.size());
933:   std::vector<double>         mass(pdgcodes.size(), 0.0);
934:   for (const auto &i : indices(pdgcodes)) {
935:     p[i]    = lts.PDG.FindByPDG(pdgcodes[i]);
936:     mass[i] = p[i].mass;
937:   }
938:
939:   // Products 4-momenta
940:   std::vector<M4Vec> p4;
941:
942:   // Do the 2 or 3-body isotropic decay
943:   if (mass.size() == 2) { gra::kinematics::TwoBodyPhaseSpace(nstar, nstar.M(), mass, p4, random); }
944:   if (mass.size() == 3) {
945:     const bool UNWEIGHT = true;
946:     gra::kinematics::ThreeBodyPhaseSpace(nstar, nstar.M(), mass, p4, UNWEIGHT, random);
947:   }
948:
949:   // Create decaytree
950:   BranchForwardSystem(p4, p, nstar, forward);
951:
952:   return true;
953: }
954:
955: // This is used with forward excitation
956: //
957: void MProcess::BranchForwardSystem(const std::vector<M4Vec> &p4, const std::vector<MParticle> &p,
958:                                    const M4Vec &nstar, gra::MDecayBranch &forward) {
959:   /*
960:    std::vector<bool> isstable(N, false);
961:    MFragment::GetDecayStatus(pdgcode, isstable);
962:   */
963:   // Construct decaytree
964:   forward   = gra::MDecayBranch();  // Initialize!!
965:   forward.p4 = nstar;
966:
967:   // Daughters
968:   forward.legs.resize(p.size());
969:   forward.depth = 0;
970:
971:   for (const auto &i : indices(p)) {
972:     // Decay particle
973:     gra::MDecayBranch branch;
974:     branch.p  = p[i];
975:     branch.p4 = p4[i];
976:
977:     // Treat pi0 -> yy (BR ~ 100%)
978:     if (branch.p.pdg == PDG::PDG_pi0) {
979:       // Decay
980:       const std::vector<double> md = {0.0, 0.0};
981:       std::vector<M4Vec>        pd;
982:       gra::kinematics::TwoBodyPhaseSpace(branch.p4, branch.p4.M(), md, pd, random);
983:
984:       // Add gamma legs
985:       branch.legs.resize(2);
986:       for (std::size_t k = 0; k < 2; ++k) {
987:         gra::MDecayBranch decaybranch;
988:         decaybranch.p  = lts.PDG.FindByPDG(PDG::PDG_gamma);
989:         decaybranch.p4 = pd[k];
990:
991:         branch.legs[k]       = decaybranch;
992:         branch.legs[k].depth = 2;
993:       }
994:     }
995:
996:
```

```
 997:     // Treat rho0 -> pi+ pi- (BR ~ 100%)
 998:     else if (branch.p.pdg == PDG::PDG_rho0) {
 999:       // Decay
1000:       const std::vector<double> md  = {PDG::mpi, PDG::mpi};
1001:       const std::vector<int>    pdg = {PDG::PDG_pip, PDG::PDG_pim};
1002:       std::vector<M4Vec>        pd;
1003:       gra::kinematics::TwoBodyPhaseSpace(branch.p4, branch.p4.M(), md, pd, random);
1004:
1005:       // Add pion legs
1006:       branch.legs.resize(2);
1007:       for (std::size_t k = 0; k < 2; ++k) {
1008:         gra::MDecayBranch decaybranch;
1009:         decaybranch.p  = lts.PDG.FindByPDG(pdg[k]);
1010:         decaybranch.p4 = pd[k];
1011:
1012:         branch.legs[k]          = decaybranch;
1013:         branch.legs[k].depth = 2;
1014:       }
1015:     }
1016:
1017:     // Here, we could add other MAJOR resonant decays
1018:     // ...
1019:
1020:     // Add leg
1021:     forward.legs[i]       = branch;
1022:     forward.legs[i].depth = 1;
1023:   }
1024: }
1025:
1026:
1027: // Fragment forward systems in central production
1028: //
1029: bool MProcess::CEPForwardFragment() {
1030:   bool                ok = true;
1031:   const std::string pt = "exp";  // Soft exponential pt
1032:
1033:   if (lts.excite1) {
1034:     const int Q = math::sign(lts.beam1.pdg);  // Charge and baryon number
1035:     const int B = Q;
1036:
1037:     if (PARAM_NSTAR::fragment == "none") {
1038:       BranchForwardSystem({}, {}, lts.pfinal[1], lts.decayforward1);
1039:     } else if (PARAM_NSTAR::fragment == "fewbody") {
1040:       if (!ExciteNstar(lts.pfinal[1], lts.decayforward1, lts.beam1)) { ok = false; }
1041:     } else if (PARAM_NSTAR::fragment == "cylinder") {
1042:       if (!ExciteContinuum(lts.pfinal[1], lts.decayforward1, lts.pfinal[1].M2(), B, Q, pt)) {
1043:         ok = false;
1044:       }
1045:     } else {
1046:       // none
1047:     }
1048:   }
1049:
1050:   if (lts.excite2) {
1051:     const int Q = math::sign(lts.beam2.pdg);  // Charge and baryon number
1052:     const int B = Q;
1053:
1054:     if (PARAM_NSTAR::fragment == "none") {
1055:       BranchForwardSystem({}, {}, lts.pfinal[2], lts.decayforward2);
1056:     } else if (PARAM_NSTAR::fragment == "fewbody") {
1057:       if (!ExciteNstar(lts.pfinal[2], lts.decayforward2, lts.beam2)) { ok = false; }
1058:     } else if (PARAM_NSTAR::fragment == "cylinder") {
1059:       if (!ExciteContinuum(lts.pfinal[2], lts.decayforward2, lts.pfinal[2].M2(), B, Q, pt)) {
1060:         ok = false;
1061:       }
1062:     } else {
1063:       // none
1064:     }
1065:   }
1066:   return ok;
1067: }
1068:
1069: bool MProcess::VetoCuts() const {
1070:   bool ok = true;
1071:
1072:   // Veto cuts
1073:   if (vetocuts.active) {
1074:     // Forward system
1075:     FindVetoCuts(lts.decayforward1, ok);
1076:     FindVetoCuts(lts.decayforward2, ok);
1077:
1078:     // Central system
1079:     for (const auto &i : indices(lts.decaytree)) { FindVetoCuts(lts.decaytree[i], ok); }
```



```
1080:   }
1081:   return ok;
1082: }
1083:
1084: // Common cuts for MResonance and MContinuum classes
1085: bool MProcess::CommonCuts() const {
1086:   bool ok = true;
1087:
1088:   // Fiducial cuts
1089:   if (fcuts.active == true) {
1090:     // Check custom user cuts (do not substitute to kinematics_ok = UserCut...)
1091:     if (!UserCut(USERCUTS, lts)) {
1092:       return false;  // not fine
1093:     }
1094:
1095:     // Check forward system variables
1096:     if (CID != "P") {  // Collinear class does not support these
1097:
1098:       if (std::abs(lts.t1) >= fcuts.forward_t_min && std::abs(lts.t1) <= fcuts.forward_t_max &&
1099:           std::abs(lts.t2) >= fcuts.forward_t_min && std::abs(lts.t2) <= fcuts.forward_t_max) {
1100:         // fine
1101:       } else {
1102:         return false;
1103:       }
1104:
1105:       if (lts.excite1) {
1106:         if (lts.pfinal[1].M() >= fcuts.forward_M_min && lts.pfinal[1].M() <= fcuts.forward_M_max) {
1107:           // fine
1108:         } else {
1109:           return false;
1110:         }
1111:       }
1112:
1113:       if (lts.excite2) {
1114:         if (lts.pfinal[2].M() >= fcuts.forward_M_min && lts.pfinal[2].M() <= fcuts.forward_M_max) {
1115:           // fine
1116:         } else {
1117:           return false;
1118:         }
1119:       }
1120:     }
1121:
1122:     // Check system variables
1123:     if (msqrt(lts.m2) >= fcuts.M_min && msqrt(lts.m2) <= fcuts.M_max && lts.Y >= fcuts.Y_min &&
1124:         lts.Y <= fcuts.Y_max && lts.Pt >= fcuts.Pt_min && lts.Pt <= fcuts.Pt_max) {
1125:       // fine, do not touch
1126:     } else {
1127:       return false;  // not fine
1128:     }
1129:
1130:     // Check fiducial cuts of the central final state particles
1131:     for (const auto &i : indices(lts.decaytree)) { FindDecayCuts(lts.decaytree[i], ok); }
1132:   }
1133:
1134:   return ok;
1135: }
1136:
1137: // Recursive find out final daughters and check if they pass fiducial cuts
1138: void MProcess::FindDecayCuts(const gra::MDecayBranch &branch, bool &ok) const {
1139:   // We are at the end -> must be a final state
1140:   if (branch.legs.size() == 0) {
1141:     // Check cuts
1142:     if (branch.p4.Pt() >= fcuts.pt_min && branch.p4.Pt() <= fcuts.pt_max &&
1143:         branch.p4.Et() >= fcuts.Et_min && branch.p4.Et() <= fcuts.Et_max &&
1144:         branch.p4.Eta() >= fcuts.eta_min && branch.p4.Eta() <= fcuts.eta_max &&
1145:         branch.p4.Rap() >= fcuts.rap_min && branch.p4.Rap() <= fcuts.rap_max) {
1146:       // Event passes cuts
1147:     } else {
1148:       ok = false;  // does not pass
1149:     }
1150:   }
1151:
1152:   // ** RECURSION here **
1153:   for (const auto &i : indices(branch.legs)) { FindDecayCuts(branch.legs[i], ok); }
1154: }
1155:
1156: // Recursive find out final daughters and check if they trigget veto cuts
1157: void MProcess::FindVetoCuts(const gra::MDecayBranch &branch, bool &ok) const {
1158:   // We are at the end -> must be a final state
1159:   if (branch.legs.size() == 0) {
1160:     // Loop over veto domains
1161:     for (const auto &i : indices(vetocuts.cuts)) {
1162:       // Check cuts
```



```
1163:       if (branch.p4.Pt() >= vetocuts.cuts[i].pt_min && branch.p4.Pt() <= vetocuts.cuts[i].pt_max &&
1164:           branch.p4.Eta() >= vetocuts.cuts[i].eta_min &&
1165:           branch.p4.Eta() <= vetocuts.cuts[i].eta_max) {
1166:         ok = false;  // VETO, does not pass
1167:       } else {
1168:         // does not trigger veto, do nothing
1169:       }
1170:     }
1171:   }
1172:
1173:   // ** RECURSION here **
1174:   for (const auto &i : indices(branch.legs)) { FindVetoCuts(branch.legs[i], ok); }
1175: }
1176:
1177: // Recursive function to plot out the decay tree
1178: void MProcess::PrintDecayTree(const gra::MDecayBranch &branch) const {
1179:   std::string spaces(branch.depth * 2, ' ');  // Give empty space
1180:   std::string lines = spaces + "â\224\224â\224\224â\224\200â\224\200>  ";
1181:
1182:   if ((branch.depth + 1) % 3 == 0) {
1183:     std::cout << rang::fg::blue << lines << rang::fg::reset;
1184:   } else if ((branch.depth + 1) % 2 == 0) {
1185:     std::cout << rang::fg::yellow << lines << rang::fg::reset;
1186:   } else {
1187:     std::cout << rang::fg::green << lines << rang::fg::reset;
1188:   }
1189:
1190:   printf(
1191:       "M = %0.2E, W = %0.2E (GeV) [<tau> = %0.1E s] PDG = [%s%d | %s] "
1192:       "[Q=%s, J=%s] \n",
1193:       branch.p.mass, branch.p.width, branch.p.tau, branch.p.pdg > 0 ? " " : "", branch.p.pdg,
1194:       branch.p.name.c_str(), gra::aux::Charge3XtoString(branch.p.charge3X).c_str(),
1195:       gra::aux::SpinZXtoString(branch.p.spinX2).c_str());
1196:
1197:   // ** RECURSION here **
1198:   for (const auto &i : indices(branch.legs)) { PrintDecayTree(branch.legs[i]); }
1199: }
1200:
1201: // Recursive function to plot out the decay tree
1202: void MProcess::PrintPhaseSpace(const gra::MDecayBranch &branch, double &product, double &product2pi,
1203:                                int &N_final) const {
1204:   std::string spaces(branch.depth * 2, ' ');  // Give empty space
1205:   std::string lines = spaces + "|â\224\200â\224\200";
1206:
1207:   if (branch.W.Integral() > 0) {  // There is decay information
1208:     printf("%s> \t {1->%llu LIPS}:  %0.3E <- %0.3E ", lines.c_str(), branch.legs.size(),
1209:            branch.W.Integral(), branch.W.IntegralError());
1210:
1211:     if (branch.PS_active) {
1212:       std::cout << "[" << rang::fg::green << "ACTIVE   " << rang::fg::reset
1213:                 << "part of integral / (2PI)]" << std::endl;
1214:     } else {
1215:       std::cout << "[" << rang::fg::red << "INACTIVE " << rang::fg::reset
1216:                 << "part of integral <=> apply manual BR]" << std::endl;
1217:     }
1218:
1219:     product *= branch.W.Integral();
1220:     product2pi *= (2 * math::PI);
1221:
1222:     for (const auto &i : indices(branch.legs)) {
1223:       PrintPhaseSpace(branch.legs[i], product, product2pi, N_final);
1224:     }
1225:   }
1226:   if (branch.legs.size() == 0) { ++N_final; }  // Final state particle
1227: }
1228:
1229: // Recursive decay tree kinematics (called event by event from inhereting
1230: // classes)
1231: bool MProcess::ConstructDecayKinematics(gra::MDecayBranch &branch) {
1232:   // This leg has any daughters
1233:   if (branch.legs.size() != 0) {
1234:     // Generate decay product masses
1235:     std::vector<double> m(branch.legs.size(), 0.0);
1236:     while (true) {
1237:       for (const auto &i : indices(branch.legs)) {
1238:         GetOffShellMass(branch.legs[i], branch.legs[i].m_offshell);
1239:         m[i] = branch.legs[i].m_offshell;
1240:       }
1241:       // Check decay products masses are not over mother mass
1242:       if (branch.p4.M() < std::accumulate(m.rbegin(), m.rend(), 0.0)) {
1243:         continue;
1244:       } else {
1245:         break;
```



```
1246:       }
1247:     }
1248:
1249:     // For now, keep always unweighted
1250:     const bool          UNWEIGHT = true;
1251:     std::vector<M4Vec> p;
1252:
1253:     // 2-body
1254:     gra::kinematics::MCW w;
1255:     if (branch.legs.size() == 2) {
1256:       w = gra::kinematics::TwoBodyPhaseSpace(branch.p4, branch.p4.M(), m, p, random);
1257:       // 3-body
1258:     } else if (branch.legs.size() == 3) {
1259:       w = gra::kinematics::ThreeBodyPhaseSpace(branch.p4, branch.p4.M(), m, p, UNWEIGHT, random);
1260:       // N-body
1261:     } else {
1262:       w = gra::kinematics::NBodyPhaseSpace(branch.p4, branch.p4.M(), m, p, UNWEIGHT, random);
1263:     }
1264:     if (w.GetW() < 0) {
1265:       std::string str =
1266:           "MProcess::ConstructDecayKinematics: Fatal error: Weight < 0 (Check your decay tree)";
1267:       std::cout << str << std::endl;
1268:       return false;
1269:     }
1270:
1271:     // Collect weight
1272:     branch.W += w;
1273:
1274:     // Collect decay product 4-momenta
1275:     for (const auto &i : indices(branch.legs)) { branch.legs[i].p4 = p[i]; }
1276:
1277:     //  ** Now the INNER recursion **
1278:     for (const auto &i : indices(branch.legs)) {
1279:       if (!ConstructDecayKinematics(branch.legs[i])) { return false; }
1280:     }
1281:   }
1282:   return true;
1283: }
1284:
1285:
1286: // Recursively add final states to the event structure
1287: void MProcess::WriteDecayKinematics(const gra::MDecayBranch &     branch,
1288:                                     const HepMC3::GenParticlePtr &mother, HepMC3::GenEvent &evt) {
1289:   // This particle has daughters
1290:   if (branch.legs.size() > 0) {
1291:     // Create new vertex with decay 4-position
1292:     HepMC3::GenVertexPtr vertex =
1293:         std::make_shared<HepMC3::GenVertex>(gra::aux::M4Vec2HepMC3(branch.decay_position));
1294:     evt.add_vertex(vertex);
1295:
1296:     // The decaying particle
1297:     vertex->add_particle_in(mother);
1298:
1299:     // Add daughters
1300:     for (const auto &i : indices(branch.legs)) {
1301:       const int STATE = (branch.legs[i].legs.size() > 0) ? PDG::PDG_DECAY : PDG::PDG_STABLE;
1302:
1303:       // ADD HERE THE ctau > 1.0 cm definition for the status
1304:       // code [TBD]
1305:
1306:       HepMC3::GenParticlePtr particle = std::make_shared<HepMC3::GenParticle>(
1307:           gra::aux::M4Vec2HepMC3(branch.legs[i].p4), branch.legs[i].p.pdg, STATE);
1308:       vertex->add_particle_out(particle);
1309:
1310:       // ** RECURSION here **
1311:       WriteDecayKinematics(branch.legs[i], particle, evt);
1312:     }
1313:   }
1314: }
1315:
1316: // Forward excitation mass sampling
1317: void MProcess::SampleForwardMasses(std::vector<double> &mvec, const std::vector<double> &randvec) {
1318:   mvec = {lts.beam1.mass, lts.beam2.mass};
1319:
1320:   lts.excite1 = false;
1321:   lts.excite2 = false;
1322:
1323:   if (EXCITATION != 0) {
1324:     M2_f_min = pow2(1.09);  // neutron + pi+ threshold
1325:     M2_f_max = gcuts.X1_max * lts.s;
1326:
1327:     log_M2_f_min = std::log(M2_f_min);
1328:     log_M2_f_max = std::log(M2_f_max);
```

```
1329:
1330:     if (M2_f_max <= M2_f_min) {
1331:       throw std::invalid_argument(
1332:           "MProcess::SampleForwardMasses: Forward leg X1 : [max = " + std::to_string(gcuts.X1_max) +
1333:           "] gives mass = " + std::to_string(msqrt(M2_f_max)) +
1334:           " GeV, below the inelastic threshold. Increase the upper (max) bound.");
1335:     }
1336:
1337:     if (EXCITATION == 1) {  // Single
1338:
1339:       // log-change of variable
1340:       const double u = log_M2_f_min + (log_M2_f_max - log_M2_f_min) * randvec[0];
1341:       const double r = std::exp(u);
1342:
1343:       if (random.U(0, 1) < 0.5) {  // 50-50
1344:         mvec[0]     = msqrt(r);
1345:         lts.excite1 = true;
1346:       } else {
1347:         mvec[1]     = msqrt(r);
1348:         lts.excite2 = true;
1349:       }
1350:     } else if (EXCITATION == 2) {  // Double
1351:
1352:       // log-change of variable
1353:       const double u1 = log_M2_f_min + (log_M2_f_max - log_M2_f_min) * randvec[0];
1354:       const double r1 = std::exp(u1);
1355:
1356:       const double u2 = log_M2_f_min + (log_M2_f_max - log_M2_f_min) * randvec[1];
1357:       const double r2 = std::exp(u2);
1358:
1359:       mvec[0]     = msqrt(r1);
1360:       mvec[1]     = msqrt(r2);
1361:       lts.excite1 = true;
1362:       lts.excite2 = true;
1363:     }
1364:   }
1365: }
1366:
1367: // This is called last by the initialization routines
1368: // as the last step before event generation.
1369: void MProcess::SetTechnicalBoundaries(gra::GENCUT &gcuts, unsigned int EXCITATION) {
1370:   if (gcuts.forward_pt_min < 0.0) {  // Not set yet by the USER
1371:     gcuts.forward_pt_min = 0.0;
1372:   }
1373:
1374:   if (gcuts.forward_pt_max < 0.0) {  // Not set yet by the USER
1375:
1376:     if (EXCITATION == 0) {  // Elastic forward protons, default values
1377:       gcuts.forward_pt_max = 2.5;
1378:     }
1379:
1380:     else if (EXCITATION == 1) {  // Single excitation
1381:       gcuts.forward_pt_max = 50.0;
1382:     } else if (EXCITATION == 2) {  // Double excitation
1383:       gcuts.forward_pt_max = 100.0;
1384:     }
1385:   }
1386: }
1387:
1388:
1389: // Forward leg integration volume
1390: double MProcess::ForwardVolume() const {
1391:   // Forward leg phi1,phi2 volumes gives (2pi)^2
1392:   const double PHI_vol = pow2(2.0 * math::PI);
1393:
1394:   // Forward leg pt volume with log-change of variables
1395:   const double J_pt = lts.pfinal[1].Pt() * lts.pfinal[2].Pt();
1396:   const double PT_vol =
1397:       J_pt * pow2(std::log(gcuts.forward_pt_max) - std::log(gcuts.forward_pt_min + ZERO_EPS));
1398:
1399:   if (EXCITATION == 0) {
1400:     return PHI_vol * PT_vol;
1401:
1402:   } else if (EXCITATION == 1) {
1403:     // log-change of variable jacobian
1404:     // \int_a^b f(M2) dM2 = \int_{ln(a)}^{ln(b)} f(exp(u)) * exp(u) du, where u = ln(M2)
1405:     const double J_M2 = (lts.excite1) ? lts.pfinal[1].M2() : lts.pfinal[2].M2();
1406:     return PHI_vol * PT_vol * J_M2 * (log_M2_f_max - log_M2_f_min);
1407:
1408:   } else if (EXCITATION == 2) {
1409:     // log-change of variable jacobian
1410:     const double J_M2 = lts.pfinal[1].M2() * lts.pfinal[2].M2();
1411:     return PHI_vol * PT_vol * J_M2 * pow2(log_M2_f_max - log_M2_f_min);
```



```
1412:   } else {
1413:     throw std::invalid_argument("MProcess::ForwardVolume: EXCITATION variable in invalid state");
1414:   }
1415: }
1416:
1417: // Print fiducial cuts set by user
1418: void MProcess::PrintFiducialCuts() const {
1419:   gra::aux::PrintBar("-");
1420:   std::cout << std::endl;
1421:   std::cout << rang::style::bold;
1422:   std::cout << "Fiducial cuts:" << std::endl << std::endl;
1423:   std::cout << rang::style::reset;
1424:
1425:   if (fcuts.active == true) {
1426:     std::cout << "Central final states" << std::endl;
1427:     printf("- Eta    [min, max] = [%0.2f, %0.2f] \n", fcuts.eta_min, fcuts.eta_max);
1428:     printf("- Pt     [min, max] = [%0.2f, %0.2f] GeV \n", fcuts.pt_min, fcuts.pt_max);
1429:     printf("- Et     [min, max] = [%0.2f, %0.2f] GeV \n", fcuts.Et_min, fcuts.Et_max);
1430:     printf("- Rap    [min, max] = [%0.2f, %0.2f] \n", fcuts.rap_min, fcuts.rap_max);
1431:     std::cout << std::endl;
1432:
1433:     std::cout << "Central system" << std::endl;
1434:     printf("- M      [min, max] = [%0.2f, %0.2f] GeV \n", fcuts.M_min, fcuts.M_max);
1435:     printf("- Pt     [min, max] = [%0.2f, %0.2f] GeV \n", fcuts.Pt_min, fcuts.Pt_max);
1436:     printf("- Rap    [min, max] = [%0.2f, %0.2f] \n", fcuts.Y_min, fcuts.Y_max);
1437:     std::cout << std::endl;
1438:
1439:     std::cout << "Forward kinematics" << std::endl;
1440:     printf("- M      [min, max] = [%0.2f, %0.2f] GeV \n", fcuts.forward_M_min, fcuts.forward_M_max);
1441:     printf("- |t_i|  [min, max] = [%0.2f, %0.2f] GeV^2 \n", fcuts.forward_t_min,
1442:            fcuts.forward_t_max);
1443:   } else {
1444:     std::cout << "- Not active" << std::endl;
1445:   }
1446:   gra::aux::PrintBar("-");
1447:   std::cout << std::endl;
1448:   std::cout << rang::style::bold;
1449:   std::cout << "Extra custom fiducial cuts:" << std::endl << std::endl;
1450:   std::cout << rang::style::reset;
1451:   std::cout << rang::fg::red;
1452:   if (USERCUTS != 0) {
1453:     printf("- Active ID = %d (see MUserCuts.cc) \n", USERCUTS);
1454:   } else {
1455:     std::cout << "- Not active" << std::endl;
1456:   }
1457:   std::cout << rang::fg::reset;
1458:
1459:   gra::aux::PrintBar("-");
1460:   std::cout << std::endl;
1461:   std::cout << rang::style::bold;
1462:   std::cout << "VETO cuts:" << std::endl << std::endl;
1463:   std::cout << rang::style::reset;
1464:
1465:   if (vetocuts.active == true) {
1466:     for (std::size_t i = 0; i < vetocuts.cuts.size(); ++i) {
1467:       std::cout << "DOMAIN: " << i << std::endl;
1468:       printf("- Eta    [min, max] = [%0.2f, %0.2f] \n", vetocuts.cuts[i].eta_min,
1469:              vetocuts.cuts[i].eta_max);
1470:       printf("- Pt     [min, max] = [%0.2f, %0.2f] GeV \n", vetocuts.cuts[i].pt_min,
1471:              vetocuts.cuts[i].pt_max);
1472:       std::cout << std::endl;
1473:     }
1474:   } else {
1475:     std::cout << "- Not active" << std::endl;
1476:   }
1477:
1478:   gra::aux::PrintBar("-");
1479:   std::cout << std::endl;
1480:   std::cout << rang::style::bold;
1481:   std::cout << "Central system decay tree:" << std::endl << std::endl;
1482:   std::cout << rang::style::reset;
1483:   // Print out decaytree recursively
1484:   for (std::size_t i = 0; i < lts.decaytree.size(); ++i) { PrintDecayTree(lts.decaytree[i]); }
1485:
1486:   std::cout << std::endl;
1487:   printf(
1488:       "Final state symmetry factor (1/S = 1/%0.0f) applied at cross section "
1489:       "level \n",
1490:       S_factor);
1491:
1492:   gra::aux::PrintBar("-");
1493:   std::cout << std::endl;
1494: }
```



```
1495:
1496:
1497: // Build and check Lorentz scalars
1498: // Input as the number of final states
1499: bool MProcess::GetLorentzScalars(unsigned int Nf) {
1500:   // Example Nf = 4 gives 8 Lorentz scalars:
1501:   //
1502:   //        t1
1503:   // ---------------->                  lts.pfinal[1]
1504:   //          |   s13
1505:   //          |------------>            lts.pfinal[3]
1506:   //          |
1507:   // s   that,uhat       shat (s34)     lts.pfinal[0]
1508:   //          |
1509:   //          |------------>            lts.pfinal[4]
1510:   //          |   s24
1511:   // ---------------->                  lts.pfinal[2]
1512:   //        t2
1513:
1514:   // s-type Lorentz scalars -->
1515:   const int offset = 3;  // central system indexing starts from offset
1516:
1517:   // Upper right triangle
1518:   for (std::size_t i = 0; i <= Nf; ++i) {  // start at 0
1519:     for (std::size_t j = 0; j <= Nf; ++j) {
1520:       if (i < j) { lts.ss[i][j] = (lts.pfinal[i] + lts.pfinal[j]).M2(); }
1521:       if (lts.ss[i][j] < 0) { return false; }
1522:     }
1523:   }
1524:   // Copy the i<->j permutated to the left bottom triangle
1525:   // (faster than calculating twice)
1526:   for (std::size_t i = 0; i <= Nf; ++i) {  // start at 0
1527:     for (std::size_t j = i; j <= Nf; ++j) {
1528:       if (i != j) { lts.ss[j][i] = lts.ss[i][j]; }
1529:     }
1530:   }
1531:
1532:   // Sub invariants w.r.t central system
1533:   lts.s1 = lts.ss[1][0];
1534:   lts.s2 = lts.ss[2][0];
1535:   if (lts.s1 < 0 || lts.s2 < 0) { return false; }
1536:
1537:   // ------------------------------------------------------------------
1538:
1539:   // t-type Lorentz scalars -->
1540:
1541:   // Propagator vectors
1542:   lts.q1 = lts.pbeam1 - lts.pfinal[1];
1543:   lts.q2 = lts.pbeam2 - lts.pfinal[2];
1544:
1545:   lts.t1 = lts.q1.M2();
1546:   lts.t2 = lts.q2.M2();
1547:
1548:   if (lts.t1 > 0 || lts.t2 > 0) { return false; }
1549:
1550:   // For 2-body central processes
1551:   if (lts.decaytree.size() == 2) {
1552:     lts.t_hat = (lts.q1 - lts.decaytree[0].p4).M2();  // note q1 on both
1553:     lts.u_hat = (lts.q1 - lts.decaytree[1].p4).M2();  // in t and u!
1554:   }
1555:
1556:   for (std::size_t i = offset; i <= Nf; ++i) { lts.tt_1[i] = (lts.q1 - lts.pfinal[i]).M2(); }
1557:   for (std::size_t i = offset; i <= Nf; ++i) {
1558:     for (std::size_t j = offset; j <= Nf; ++j) {
1559:       lts.tt_xy[i][j] = (lts.q1 - lts.pfinal[i] - lts.pfinal[j]).M2();
1560:     }
1561:   }
1562:   for (std::size_t i = offset; i <= Nf; ++i) { lts.tt_2[i] = (lts.q2 - lts.pfinal[i]).M2(); }
1563:
1564:   // Fractional longitudinal momentum loss [0,1]
1565:   lts.x1 = (1 - lts.pfinal[1].Pz() / lts.pbeam1.Pz());
1566:   lts.x2 = (1 - lts.pfinal[2].Pz() / lts.pbeam2.Pz());
1567:
1568:   // Bjorken-x [0,1] (this Lorentz invariant expression
1569:   // gives 1 always for elastic central production forward leg)
1570:   lts.xbj1 = lts.t1 / (2 * (lts.pbeam1 * lts.q1));
1571:   lts.xbj2 = lts.t2 / (2 * (lts.pbeam2 * lts.q2));
1572:
1573:   // Propagator pt
1574:   lts.qt1 = lts.q1.Pt();
1575:   lts.qt2 = lts.q2.Pt();
1576:
1577:   // Often used system variables
```



```
1578:   lts.m2    = lts.pfinal[0].M2();
1579:   lts.s_hat = lts.m2;
1580:   lts.Y     = lts.pfinal[0].Rap();
1581:   lts.Pt    = lts.pfinal[0].Pt();
1582:
1583:   return true;
1584: }
1585:
1586: // Event output recording (to HepMC containers)
1587: //
1588: bool MProcess::CommonRecord(HepMC3::GenEvent &evt) {
1589:   // Initial states (4-momentum, pdg-id, status code)
1590:   HepMC3::GenParticlePtr gen_p1 = std::make_shared<HepMC3::GenParticle>(
1591:       gra::aux::M4Vec2HepMC3(lts.pbeam1), lts.beam1.pdg, PDG::PDG_BEAM);
1592:   HepMC3::GenParticlePtr gen_p2 = std::make_shared<HepMC3::GenParticle>(
1593:       gra::aux::M4Vec2HepMC3(lts.pbeam2), lts.beam2.pdg, PDG::PDG_BEAM);
1594:
1595:   // Final states (protons/excited system)
1596:   int PDG_ID1 = lts.beam1.pdg;
1597:   int PDG_ID2 = lts.beam2.pdg;
1598:
1599:   int PDG_status1 = PDG::PDG_STABLE;
1600:   int PDG_status2 = PDG::PDG_STABLE;
1601:
1602:   if (lts.excite1 == true) {
1603:     PDG_ID1     = std::abs(PDG::PDG_NSTAR) * math::sign(lts.beam1.pdg);
1604:     PDG_status1 = PDG::PDG_INTERMEDIATE;
1605:   }
1606:   if (lts.excite2 == true) {
1607:     PDG_ID2     = std::abs(PDG::PDG_NSTAR) * math::sign(lts.beam2.pdg);
1608:     PDG_status2 = PDG::PDG_INTERMEDIATE;
1609:   }
1610:
1611:   HepMC3::GenParticlePtr gen_p1f = std::make_shared<HepMC3::GenParticle>(
1612:       gra::aux::M4Vec2HepMC3(lts.pfinal[1]), PDG_ID1, PDG_status1);
1613:   HepMC3::GenParticlePtr gen_p2f = std::make_shared<HepMC3::GenParticle>(
1614:       gra::aux::M4Vec2HepMC3(lts.pfinal[2]), PDG_ID2, PDG_status2);
1615:
1616:   // ------------------------------------------------------------------
1617:
1618:   // Propagator 1 and 2
1619:   // it is ill-posed to try classify pomeron/gamma/gluon etc. here, thus,
1620:   // we tag only a generic propagator ID in the record
1621:   HepMC3::GenParticlePtr gen_q1 = std::make_shared<HepMC3::GenParticle>(
1622:       gra::aux::M4Vec2HepMC3(lts.q1), PDG::PDG_propagator, PDG::PDG_INTERMEDIATE);
1623:   HepMC3::GenParticlePtr gen_q2 = std::make_shared<HepMC3::GenParticle>(
1624:       gra::aux::M4Vec2HepMC3(lts.q2), PDG::PDG_propagator, PDG::PDG_INTERMEDIATE);
1625:
1626:   // ------------------------------------------------------------------
1627:
1628:   // Central system / resonance
1629:   HepMC3::GenParticlePtr gen_q = std::make_shared<HepMC3::GenParticle>(
1630:       gra::aux::M4Vec2HepMC3(lts.pfinal[0]), PDG::PDG_system, PDG::PDG_INTERMEDIATE);
1631:
1632:   // ==================================================================
1633:   // Construct vertices
1634:
1635:   // Upper proton-propagator-proton
1636:   HepMC3::GenVertexPtr v1 = std::make_shared<HepMC3::GenVertex>();
1637:   v1->add_particle_in(gen_p1);
1638:   v1->add_particle_out(gen_p1f);
1639:   v1->add_particle_out(gen_q1);
1640:
1641:   // Lower proton-propagator-proton
1642:   HepMC3::GenVertexPtr v2 = std::make_shared<HepMC3::GenVertex>();
1643:   v2->add_particle_in(gen_p2);
1644:   v2->add_particle_out(gen_p2f);
1645:   v2->add_particle_out(gen_q2);
1646:
1647:   // Propagator-Propagator-System vertex
1648:   HepMC3::GenVertexPtr v3 = std::make_shared<HepMC3::GenVertex>();
1649:   v3->add_particle_in(gen_q1);
1650:   v3->add_particle_in(gen_q2);
1651:   v3->add_particle_out(gen_q);
1652:
1653:   evt.add_vertex(v1);
1654:   evt.add_vertex(v2);
1655:   evt.add_vertex(v3);
1656:
1657:   // ==================================================================
1658:   // System->Decay products vertex
1659:
1660:   HepMC3::GenVertexPtr v4 = std::make_shared<HepMC3::GenVertex>();
```

./src/MProcess.cc 21/23

```
1661:   evt.add_vertex(v4);
1662:
1663:   // Add resonance in
1664:   v4->add_particle_in(gen_q);
1665:
1666:   // Add direct daughters
1667:   for (const auto &i : indices(lts.decaytree)) {
1668:     const int STATE = (lts.decaytree[i].legs.size() > 0) ? PDG::PDG_DECAY : PDG::PDG_STABLE;
1669:
1670:     // TBD: ADD HERE THE ctau > 1.0 cm definition for the status code
1671:
1672:     if (STATE == PDG::PDG_DECAY) {
1673:       // ----------------------------------------------------------
1674:       // Pick exponential lifetime in the rest frame
1675:       const double tau = random.ExpRandom(1.0 / lts.decaytree[i].p.tau);
1676:
1677:       // Get decay vertex coordinates in the lab
1678:       const double MM                = 1E3;  // meters to millimeters
1679:       lts.decaytree[i].decay_position = lts.decaytree[i].p4.PropagateAPosition(tau, MM);
1680:       // ----------------------------------------------------------
1681:     }
1682:
1683:     HepMC3::GenParticlePtr particle = std::make_shared<HepMC3::GenParticle>(
1684:         gra::aux::M4Vec2HepMC3(lts.decaytree[i].p4), lts.decaytree[i].p.pdg, STATE);
1685:     v4->add_particle_out(particle);
1686:
1687:     WriteDecayKinematics(lts.decaytree[i], particle, evt);
1688:   }
1689:
1690:   // ------------------------------------------------------------------
1691:   // Crucial, check do we need to fragment
1692:   // with vetocuts INactive, fragmentation done at last step here (for efficiency)
1693:   if (!CEPForwardFragment()) { return false; }
1694:
1695:   // Upper proton excitation
1696:   if (lts.excite1 && lts.decayforward1.legs.size() != 0) {
1697:     SaveBranch(evt, lts.decayforward1, gen_p1f);
1698:   }
1699:
1700:   // Lower proton excitation
1701:   if (lts.excite2 && lts.decayforward2.legs.size() != 0) {
1702:     SaveBranch(evt, lts.decayforward2, gen_p2f);
1703:   }
1704:
1705:   return true;
1706: }
1707:
1708: // Save branch to event with mother pX
1709: void MProcess::SaveBranch(HepMC3::GenEvent &evt, const gra::MDecayBranch &branch,
1710:                           const HepMC3::GenParticlePtr &pX) {
1711:   // Create vertex
1712:   HepMC3::GenVertexPtr vX = std::make_shared<HepMC3::GenVertex>();
1713:   evt.add_vertex(vX);
1714:
1715:   // Add mother in
1716:   vX->add_particle_in(pX);
1717:
1718:   // Add direct daughters
1719:   for (const auto &i : indices(branch.legs)) {
1720:     const int STATE = (branch.legs[i].legs.size() > 0) ? PDG::PDG_DECAY : PDG::PDG_STABLE;
1721:     // TBD: ADD HERE THE ctau > 1.0 cm definition for the status code
1722:
1723:     HepMC3::GenParticlePtr particle = std::make_shared<HepMC3::GenParticle>(
1724:         gra::aux::M4Vec2HepMC3(branch.legs[i].p4), branch.legs[i].p.pdg, STATE);
1725:     vX->add_particle_out(particle);
1726:
1727:     WriteDecayKinematics(branch.legs[i], particle, evt);
1728:   }
1729: }
1730:
1731: // Set process
1732: void MProcess::SetProcess(std::string &str, const std::vector<aux::OneCMD> &syntax) {
1733:   // SET IT HERE!
1734:   PROCESS = str;
1735:
1736:   // Call this always first
1737:   lts.PDG.ReadParticleData(gra::aux::GetBasePath(2) + "/modeldata/mass_width_2018.mcd");
1738:
1739:   // @SYNTAX Read and set new PDG input
1740:   for (const auto &i : indices(syntax)) {
1741:     if (syntax[i].id == "PDG") {
1742:       // Take target string
1743:       std::string pdg_;
```

./src/MProcess.cc 22/23

```
1744:       if (syntax[i].target.size() == 1) {
1745:         pdg_ = syntax[i].target[0];
1746:       } else if (syntax[i].target.size() == 0) {
1747:         throw std::invalid_argument("@Syntax error: invalid PDG[] without any target []");
1748:       } else if (syntax[i].target.size() > 1) {
1749:         throw std::invalid_argument("@Syntax error: invalid PDG[] with multiple targets inside []");
1750:       }
1751:       // Conversion to integer
1752:       int pdg = 0;
1753:
1754:       try {
1755:         pdg = std::stoi(pdg_);
1756:       } catch (...) {
1757:         throw std::invalid_argument(
1758:             "@Syntax error: invalid PDG[] target number, string to int conversion fails");
1759:       }
1760:
1761:       // Try to find the particle from PDG table, will throw exception if fails
1762:       MParticle p = lts.PDG.FindByPDG(pdg);
1763:
1764:       // Check if it has an anti-particle
1765:       MParticle p_anti;
1766:       bool      found_anti = false;
1767:       try {
1768:         p_anti     = lts.PDG.FindByPDG(-pdg);
1769:         found_anti = true;
1770:       } catch (...) {}  // do nothing,
1771:
1772:       // Set new properties
1773:       for (const auto &x : syntax[i].arg) {
1774:         if (x.first == "M") {
1775:           p.mass = std::stod(x.second);
1776:           if (found_anti) { p_anti.mass = p.mass; }
1777:         }
1778:         if (x.first == "W") {
1779:           p.width = std::stod(x.second);
1780:           p.tau   = PDG::hbar / p.width;  // mean lifetime in the rest frame
1781:           if (found_anti) {
1782:             p_anti.width = p.width;
1783:             p_anti.tau   = p.tau;
1784:           }
1785:         }
1786:       }
1787:       // Set new modified to the PDG table
1788:       lts.PDG.PDG_table[pdg] = p;
1789:       if (found_anti) { lts.PDG.PDG_table[-pdg] = p_anti; }
1790:
1791:       std::cout << rang::fg::red
1792:                 << "MProcess::SetProcess: New particle properties set "
1793:                   "with @PDG[number]{key:val} syntax:"
1794:                 << rang::fg::reset << std::endl;
1795:       p.print();
1796:     }
1797:   }
1798:
1799:   // Remove whitespace
1800:   std::string::iterator end_pos = std::remove(str.begin(), str.end(), ' ');
1801:   str.erase(end_pos, str.end());
1802:
1803:   // Parse commandline string
1804:   std::string istate  = "";
1805:   std::string channel = "";
1806:   std::string mc      = "";
1807:   ParseCMD(str, istate, channel, mc);
1808:
1809:   // Setup subprocess
1810:   ProcPtr.Initialize(istate, channel);
1811:
1812:   // Check do we find the process
1813:   if (ProcPtr.Processss.count(str)) {
1814:     // fine
1815:   } else {
1816:     throw std::invalid_argument("MProcess::SetProcess: Unknown PROCESS: " + str);
1817:   }
1818: }
1819:
1820: // "PP[RES]<C>" is an example of valid string to be parsed
1821: //
1822: void MProcess::ParseCMD(const std::string &str, std::string &first, std::string &second,
1823:                         std::string &third) const {
1824:   // First and Second
1825:   bool found = false;
1826:   ;
```

./src/MProcess.cc 23/23

```
1827:   for (const auto &i : indices(str)) {
1828:     if (str[i] != '[' && !found) {
1829:       first += str[i];
1830:       continue;
1831:     } else if (str[i] == '[' && !found) {
1832:       found = true;
1833:       continue;
1834:     }
1835:     if (found && str[i] != ']') { second += str[i]; }
1836:     if (found && str[i] == ']') { break; }
1837:   }
1838:
1839:   // Third
1840:   int mark1 = 0;
1841:   int mark2 = 0;
1842:   for (const auto &i : indices(str)) {
1843:     if (str[i] == '<') {
1844:       mark1 = i;
1845:       continue;
1846:     }
1847:     if (str[i] == '>') {
1848:       mark2 = i;
1849:       break;  // First >, then break
1850:     }
1851:   }
1852:   third = str.substr(mark1 + 1, mark2 - mark1 - 1);
1853: }
1854:
1855:
1856: }  // namespace gra
1857:
```

./src/MQuasiElastic.cc 1/11

```
  1: // QuasiElastic (EL,SD,DD) and soft ND (simplified) class
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: // C++
  7: #include <complex>
  8: #include <iostream>
  9: #include <random>
 10: #include <vector>
 11:
 12: // HepMC3
 13: #include "HepMC3/FourVector.h"
 14: #include "HepMC3/GenEvent.h"
 15: #include "HepMC3/GenParticle.h"
 16: #include "HepMC3/GenVertex.h"
 17:
 18: // Own
 19: #include "Graniitti/MAux.h"
 20: #include "Graniitti/MForm.h"
 21: #include "Graniitti/MKinematics.h"
 22: #include "Graniitti/MMath.h"
 23: #include "Graniitti/MProcess.h"
 24: #include "Graniitti/MQuasiElastic.h"
 25: #include "Graniitti/MUserCuts.h"
 26:
 27: using gra::PDG::GeV2barn;
 28: using gra::PDG::mp;
 29: using gra::aux::indices;
 30: using gra::math::PI;
 31: using gra::math::abs2;
 32: using gra::math::msqrt;
 33: using gra::math::pow2;
 34: using gra::math::zi;
 35:
 36: namespace gra {
 37: // This is needed by construction
 38: MQuasiElastic::MQuasiElastic() { Initialize(); }
 39:
 40: // Constructor
 41: MQuasiElastic::MQuasiElastic(std::string process, const std::vector<aux::OneCMD> &syntax) {
 42:   Initialize();
 43:   InitHistograms();
 44:   SetProcess(process, syntax);
 45:
 46:   // Init final states
 47:   M4Vec zerovec(0, 0, 0, 0);
 48:   for (std::size_t i = 0; i < 10; ++i) { lts.pfinal.push_back(zerovec); }
 49:
 50:   // Pomeron weights
 51:   MAXPOMW = std::vector<double>(100, 0.0);
 52:   std::cout << "MQuasiElastic:: [Constructor done]" << std::endl;
 53: }
 54:
 55: void MQuasiElastic::Initialize() {
 56:   const std::vector<std::string> supported = {"X"};
 57:   CID                                      = "Q";
 58:   ProcPtr                                  = MSubProc(supported, CID);
 59: }
 60:
 61: // Destructor
 62: MQuasiElastic::~MQuasiElastic() {}
 63:
 64: // Initialize cut and process spesific postsetup
 65: void MQuasiElastic::post_Constructor() {
 66:   // Set phase space dimension
 67:   if (ProcPtr.CHANNEL == "EL") { ProcPtr.LIPSDIM = 1; };
 68:   if (ProcPtr.CHANNEL == "SD") { ProcPtr.LIPSDIM = 2; };
 69:   if (ProcPtr.CHANNEL == "DD") { ProcPtr.LIPSDIM = 3; };
 70:   if (ProcPtr.CHANNEL == "ND") { ProcPtr.LIPSDIM = 1; };  // Keep it 1
 71:
 72:   Eikonal.Numerics.MaxLoopKT = 3.0;
 73: }
 74:
 75: // Fiducial user cuts
 76: bool MQuasiElastic::FiducialCuts() const {
 77:   if (fcuts.active == true) {
 78:     // EL cuts
 79:     if (ProcPtr.CHANNEL == "EL") {
 80:       if (fcuts.forward_t_min < std::abs(lts.t) && std::abs(lts.t) < fcuts.forward_t_max) {
 81:         // fine
 82:       } else {
 83:         return false;  // not fine
```

```
84:       }
85:     }
86:
87:     // SD cuts
88:     if (ProcPtr.CHANNEL == "SD") {
89:       if (lts.pfinal[1].M() > 1.0) {  // this one is excited system
90:         if (fcuts.forward_M_min < lts.pfinal[1].M() && lts.pfinal[1].M() < fcuts.forward_M_max &&
91:             fcuts.forward_t_min < std::abs(lts.t) && std::abs(lts.t) < fcuts.forward_t_max) {
92:           // fine
93:         } else {
94:           return false;  // not fine
95:         }
96:       } else {
97:         if (fcuts.forward_M_min < lts.pfinal[2].M() && lts.pfinal[2].M() < fcuts.forward_M_max &&
98:             fcuts.forward_t_min < std::abs(lts.t) && std::abs(lts.t) < fcuts.forward_t_max) {
99:           // fine
100:        } else {
101:          return false;  // not fine
102:        }
103:      }
104:    }
105:
106:    // DD cuts
107:    if (ProcPtr.CHANNEL == "DD") {
108:      if (fcuts.forward_M_min < lts.pfinal[1].M() && lts.pfinal[1].M() < fcuts.forward_M_max &&
109:          fcuts.forward_M_min < lts.pfinal[2].M() && lts.pfinal[2].M() < fcuts.forward_M_max &&
110:          fcuts.forward_t_min < std::abs(lts.t) && std::abs(lts.t) < fcuts.forward_t_max) {
111:        // fine
112:      } else {
113:        return false;  // not fine
114:      }
115:    }
116:
117:    // Check user cuts (do not substitute to kinematics =
118:    // UserCut...)
119:    if (!UserCut(USERCUTS, lts)) {
120:      return false;  // not fine
121:    }
122:  }
123:  return true;  // fine
124: }
125:
126: bool MQuasiElastic::LoopKinematics(const std::vector<double> &p1p, const std::vector<double> &p2p) {
127:   static const M4Vec beamsum = lts.pbeam1 + lts.pbeam2;
128:
129:   const double m1 = lts.pfinal[1].M();
130:   const double m2 = lts.pfinal[2].M();
131:
132:   // SET new proton pT degrees of freedom
133:   lts.pfinal[1].SetPxPy(p1p[0], p1p[1]);
134:   lts.pfinal[2].SetPxPy(p2p[0], p2p[1]);
135:
136:   // SOLVE pz and E
137:   const double p1z =
138:       kinematics::SolvePz(m1, m2, lts.pfinal[1].Pt(), lts.pfinal[2].Pt(), 0, 0, lts.s);
139:   const double p2z = -p1z;  // by momentum conservation
140:
141:   // pz and E of protons/N*
142:   lts.pfinal[1].SetPzE(p1z, msqrt(pow2(m1) + pow2(lts.pfinal[1].Pt()) + pow2(p1z)));
143:   lts.pfinal[2].SetPzE(p2z, msqrt(pow2(m2) + pow2(lts.pfinal[2].Pt()) + pow2(p2z)));
144:
145:   // ------------------------------------------------------------
146:   // Now boost if asymmetric beams
147:   if (std::abs(beamsum.Pz()) > 1e-6) {
148:     constexpr int sign = 1;  // positive -> boost to the lab
149:     kinematics::LorentzBoost(beamsum, lts.sqrt_s, lts.pfinal[1], sign);
150:     kinematics::LorentzBoost(beamsum, lts.sqrt_s, lts.pfinal[2], sign);
151:   }
152:   // ------------------------------------------------------------
153:
154:   if (!gra::math::CheckEMC(beamsum - (lts.pfinal[1] + lts.pfinal[2]))) { return false; }
155:
156:   return B3GetLorentzScalars();
157: }
158:
159: // Get weight
160: double MQuasiElastic::EventWeight(const std::vector<double> &randvec, AuxIntData &aux) {
161:   double W = 0.0;
162:
163:   if (ProcPtr.CHANNEL != "ND") {  // Diffractive
164:
165:     aux.kinematics_ok = B3RandomKin(randvec);
166:     aux.fidcuts_ok    = FiducialCuts();
```



```
167:     aux.vetocuts_ok   = VetoCuts();
168:
169:     if (aux.Valid()) {
170:       // ** EVENT WEIGHT **
171:       const double LIPS   = B3PhaseSpaceWeight();  // Phase-space weight
172:       const double MatESQ = GetAmp2();             // Matrix element squared
173:
174:       // Total weight: phase-space x |M|^2 x barn units
175:       W = LIPS * B3IntegralVolume() * MatESQ * GeV2barn;
176:     }
177:
178:     aux.amplitude_ok = CheckInfNan(W);
179:
180:     // Fill Histograms
181:     // const double totalweight = W * aux.vegasweight;
182:     // FillHistograms(totalweight, lts); // This seqfaults, because we do not
183:     // have
184:     // enough observables
185:   } else {  // Non-Diffractive
186:
187:     aux.kinematics_ok = true;  // This is always the case
188:     aux.fidcuts_ok    = true;
189:     aux.vetocuts_ok   = true;
190:
191:     const double LIPS   = B3PhaseSpaceWeight();     // Phase-space weight
192:     const double MatESQ = abs2(PolySoft(randvec));  // Matrix element squared
193:
194:     // W = LIPS * MatESQ * GeV2barn;  // Total weight: phase-space x |M|^2 x barn units
195:     W = LIPS * MatESQ;  // Total weight:
196:
197:     aux.amplitude_ok = CheckInfNan(W);
198:
199:     // Trigger forced event generation (using last component of auxvar)
200:     // (no need for outside acceptance-rejection with this process)
201:     if (W > 0) {
202:       aux.forced_accept = true;
203:     } else {
204:       aux.forced_accept = false;  // Failed event
205:     }
206:   }
207:
208:   return W;
209: }
210:
211: // Record HepMC3 event
212: bool MQuasiElastic::EventRecord(HepMC3::GenEvent &evt) {
213:   // Non-Diffractive
214:   if (ProcPtr.CHANNEL == "ND") {
215:     HepMC3::GenParticlePtr gen_p1;
216:     HepMC3::GenParticlePtr gen_p2;
217:     HepMC3::GenParticlePtr gen_p1f;
218:     HepMC3::GenParticlePtr gen_p2f;
219:
220:     // Loop over multiple "cut pomerons"
221:     for (const auto &i : indices(etree)) {
222:       if (i == 0) {  // First
223:
224:         // Initial state protons (4-momentum, pdg-id, status code)
225:         gen_p1 = std::make_shared<HepMC3::GenParticle>(
226:             gra::aux::M4Vec2HepMC3(etree[i].p1i), (i == 0) ? lts.beam1.pdg : PDG::PDG_fragment,
227:             (i == 0) ? PDG::PDG_BEAM : PDG::PDG_INTERMEDIATE);
228:
229:         gen_p2 = std::make_shared<HepMC3::GenParticle>(
230:             gra::aux::M4Vec2HepMC3(etree[i].p2i), (i == 0) ? lts.beam2.pdg : PDG::PDG_fragment,
231:             (i == 0) ? PDG::PDG_BEAM : PDG::PDG_INTERMEDIATE);
232:
233:       } else {  // Others recursively
234:
235:         gen_p1 = gen_p1f;
236:         gen_p2 = gen_p2f;
237:       }
238:
239:       // Final state (or intermediate in the chain) fragments
240:       gen_p1f = std::make_shared<HepMC3::GenParticle>(gra::aux::M4Vec2HepMC3(etree[i].p1f),
241:                                                       PDG::PDG_fragment, PDG::PDG_INTERMEDIATE);
242:
243:       gen_p2f = std::make_shared<HepMC3::GenParticle>(gra::aux::M4Vec2HepMC3(etree[i].p2f),
244:                                                       PDG::PDG_fragment, PDG::PDG_INTERMEDIATE);
245:
246:       // Exchange objects
247:       HepMC3::GenParticlePtr gen_q1 = std::make_shared<HepMC3::GenParticle>(
248:           gra::aux::M4Vec2HepMC3(etree[i].q1), PDG::PDG_propagator, PDG::PDG_INTERMEDIATE);
249:       HepMC3::GenParticlePtr gen_q2 = std::make_shared<HepMC3::GenParticle>(
```



```
250:           gra::aux::M4Vec2HepMC3(etree[i].q2), PDG::PDG_propagator, PDG::PDG_INTERMEDIATE);
251:
252:       // Virtual state
253:       M4Vec system4vec(etree[i].k.Px(), etree[i].k.Py(), etree[i].k.Pz(), etree[i].k.E());
254:
255:       HepMC3::GenParticlePtr gen_X = std::make_shared<HepMC3::GenParticle>(
256:           gra::aux::M4Vec2HepMC3(system4vec), PDG::PDG_system, PDG::PDG_INTERMEDIATE);
257:
258:       HepMC3::GenVertexPtr vX = std::make_shared<HepMC3::GenVertex>();
259:
260:       if (i != etree.size() - 1) {
261:         // Upper vertex
262:         HepMC3::GenVertexPtr vXUP = std::make_shared<HepMC3::GenVertex>();
263:         vXUP->add_particle_in(gen_p1);
264:         vXUP->add_particle_out(gen_p1f);
265:         vXUP->add_particle_out(gen_q1);
266:
267:         // Lower vertex
268:         HepMC3::GenVertexPtr vXDO = std::make_shared<HepMC3::GenVertex>();
269:         vXDO->add_particle_in(gen_p2);
270:         vXDO->add_particle_out(gen_p2f);
271:         vXDO->add_particle_out(gen_q2);
272:
273:         // Add vertices
274:         evt.add_vertex(vXUP);
275:         evt.add_vertex(vXDO);
276:
277:         // Pomeron-Pomeron-Virtual state vertex
278:         vX->add_particle_in(gen_q1);
279:         vX->add_particle_in(gen_q2);
280:         vX->add_particle_out(gen_X);
281:
282:       } else {  // Last MPI, fuse remnant + remnant -> system
283:
284:         // Proton-Proton-Virtual state vertex
285:         vX = std::make_shared<HepMC3::GenVertex>();
286:         vX->add_particle_in(gen_p1);
287:         vX->add_particle_in(gen_p2);
288:         vX->add_particle_out(gen_X);
289:       }
290:
291:       // Add vertex to the event
292:       evt.add_vertex(vX);
293:
294:       // ------------------------------------------------------------
295:       // Quantum numbers distributed here
296:       int    B      = 0;  // Baryon number
297:       int    Q      = 0;  // Charge
298:       double Q2_scale = system4vec.M2();
299:
300:       // Beam fragments carry the initial state quantum numbers,
301:       // we distribute them here for the second last and the last MPI
302:       if (i == etree.size() - 2) {
303:         B = math::sign(lts.beam2.pdg);  // Proton(anti) proton beams only
304:         Q = B;
305:       }
306:       if (i == etree.size() - 1) {
307:         B = math::sign(lts.beam1.pdg);
308:         Q = B;
309:       }
310:
311:       // Excite it
312:       MDecayBranch branch;
313:       if (!ExciteContinuum(system4vec, branch, Q2_scale, B, Q)) {
314:         std::cout << "ND failed with ExciteContinuum" << std::endl;
315:         return false;  // failed
316:       }
317:
318:       // Save it
319:       SaveBranch(evt, branch, gen_X);
320:     }
321:
322:     return true;
323:   }
324:
325:   // Diffractive processes
326:
327:   // Initial states (4-momentum, pdg-id, status code)
328:   HepMC3::GenParticlePtr gen_p1 = std::make_shared<HepMC3::GenParticle>(
329:       gra::aux::M4Vec2HepMC3(lts.pbeam1), lts.beam1.pdg, PDG::PDG_BEAM);
330:   HepMC3::GenParticlePtr gen_p2 = std::make_shared<HepMC3::GenParticle>(
331:       gra::aux::M4Vec2HepMC3(lts.pbeam2), lts.beam2.pdg, PDG::PDG_BEAM);
332:
```



```
333:   // Propagator 4-vector and generator particle
334:   M4Vec    q1(lts.pbeam1 - lts.pfinal[1]);
335:   HepMC3::GenParticlePtr gen_q1 = std::make_shared<HepMC3::GenParticle>(
336:       gra::aux::M4Vec2HepMC3(q1), PDG::PDG_propagator, PDG::PDG_INTERMEDIATE);
337:
338:   // Final state protons/excited systems
339:   int PDG_ID1     = lts.beam1.pdg;
340:   int PDG_ID2     = lts.beam2.pdg;
341:   int PDG_status1 = PDG::PDG_STABLE;
342:   int PDG_status2 = PDG::PDG_STABLE;
343:
344:   // EL
345:   // already fine
346:
347:   // SD
348:   if (ProcPtr.CHANNEL == "SD") {
349:     if (lts.excite1) {  // proton 1 excited
350:       PDG_ID1     = std::abs(PDG::PDG_NSTAR) * math::sign(lts.beam1.pdg);
351:       PDG_status1 = PDG::PDG_INTERMEDIATE;
352:     } else {
353:       PDG_ID2     = std::abs(PDG::PDG_NSTAR) * math::sign(lts.beam2.pdg);
354:       PDG_status2 = PDG::PDG_INTERMEDIATE;
355:     }
356:   }
357:   // DD
358:   if (ProcPtr.CHANNEL == "DD") {
359:     PDG_ID1     = std::abs(PDG::PDG_NSTAR) * math::sign(lts.beam1.pdg);
360:     PDG_status1 = PDG::PDG_INTERMEDIATE;
361:     PDG_ID2     = std::abs(PDG::PDG_NSTAR) * math::sign(lts.beam2.pdg);
362:     PDG_status2 = PDG::PDG_INTERMEDIATE;
363:   }
364:
365:   HepMC3::GenParticlePtr gen_p1f = std::make_shared<HepMC3::GenParticle>(
366:       gra::aux::M4Vec2HepMC3(lts.pfinal[1]), PDG_ID1, PDG_status1);
367:   HepMC3::GenParticlePtr gen_p2f = std::make_shared<HepMC3::GenParticle>(
368:       gra::aux::M4Vec2HepMC3(lts.pfinal[2]), PDG_ID2, PDG_status2);
369:
370:   // Construct vertices
371:
372:   // Upper proton-pomeron-proton
373:   HepMC3::GenVertexPtr v1 = std::make_shared<HepMC3::GenVertex>();
374:   v1->add_particle_in(gen_p1);
375:   v1->add_particle_out(gen_p1f);
376:   v1->add_particle_out(gen_q1);
377:
378:   // Lower proton-pomeron-proton
379:   HepMC3::GenVertexPtr v2 = std::make_shared<HepMC3::GenVertex>();
380:   v2->add_particle_in(gen_p2);
381:   v2->add_particle_out(gen_p2f);
382:   v2->add_particle_in(gen_q1);
383:
384:   // Finally add all vertices
385:   evt.add_vertex(v1);
386:   evt.add_vertex(v2);
387:
388:   // Upper proton excitation
389:   if (lts.excite1) {
390:     const int B = math::sign(lts.beam1.pdg);  // baryon number the same as charge
391:     const int Q = B;
392:
393:     if (!ExciteContinuum(lts.pfinal[1], lts.decayforward1, lts.pfinal[1].M2(), B, Q)) {
394:       return false;  // failed
395:     }
396:     SaveBranch(evt, lts.decayforward1, gen_p1f);
397:   }
398:   // Lower proton excitation
399:   if (lts.excite2) {
400:     const int B = math::sign(lts.beam2.pdg);  // baryon number the same as charge
401:     const int Q = B;
402:
403:     if (!ExciteContinuum(lts.pfinal[2], lts.decayforward2, lts.pfinal[2].M2(), B, Q)) {
404:       return false;  // failed
405:     }
406:     SaveBranch(evt, lts.decayforward2, gen_p2f);
407:   }
408:   return true;
409: }
410:
411:
412: // Print out setup
413: void MQuasiElastic::PrintInit(bool silent) const {
414:   if (!silent) {
415:     PrintSetup();
```

```
416:
417:     // Diffractive processes
418:     if (ProcPtr.CHANNEL != "ND") {
419:       std::string proton1 = "-----------EL---------->";
420:       std::string proton2 = "-----------EL---------->";
421:
422:       if (ProcPtr.CHANNEL == "SD") { proton1 = "-----------F2-xxxxxxxx>"; }
423:       if (ProcPtr.CHANNEL == "DD") {
424:         proton1 = "-----------F2-xxxxxxxx>";
425:         proton2 = "-----------F2-xxxxxxxx>";
426:       }
427:
428:       std::vector<std::string> feynmangraph;
429:       feynmangraph = {"||          ", "||          ", "||          ",
430:                       "||          "};
431:
432:       // Print diagram
433:       std::cout << proton1 << std::endl;
434:       for (const auto &i : indices(feynmangraph)) {
435:         if (SCREENING) {  // Put red
436:           std::cout << rang::fg::red << "     **     " << rang::style::reset;
437:         } else {
438:           std::cout << rang::fg::red << "            " << rang::style::reset;
439:         }
440:         std::cout << feynmangraph[i] << std::endl;
441:       }
442:       std::cout << proton2 << std::endl << std::endl;
443:
444:       // Generation cuts
445:       std::cout << rang::style::bold << "Generation cuts:" << rang::style::reset << std::endl
446:                 << std::endl;
447:
448:       if (ProcPtr.CHANNEL != "EL") {
449:         printf("~ X1  [min, max]   = [%0.3E, %0.3E] (X1 == M^2/s) \n", gcuts.X1_min, gcuts.X1_max);
450:       }
451:       printf("~ |t| [max]        = %0.3f GeV^2 \n", pow2(Eikonal.Numerics.MaxLoopKT));
452:       std::cout << std::endl;
453:
454:       // Fiducial cuts
455:       std::cout << rang::style::bold << "Fiducial cuts:" << rang::style::reset << std::endl
456:                 << std::endl;
457:       if (fcuts.active) {
458:         printf("~ M   [min, max]   = [%0.2f, %0.2f] GeV^2 \n", fcuts.forward_M_min,
459:                fcuts.forward_M_max);
460:         printf("~ |t| [min, max]   = [%0.2f, %0.2f] GeV^2 \n", fcuts.forward_t_min,
461:                fcuts.forward_t_max);
462:         std::cout << std::endl;
463:       } else {
464:         std::cout << "- Not active" << std::endl;
465:       }
466:
467:       // Non-Diffractive
468:     } else {
469:       for (std::size_t i = 0; i < 7; ++i) {
470:         if (i > 0 && i < 6) {
471:           if (SCREENING) {
472:             std::cout << rang::fg::red << "     **     " << rang::style::reset;
473:           } else {
474:             std::cout << rang::fg::red << "            " << rang::style::reset;
475:           }
476:         } else {
477:           std::cout << "-----------";
478:         }
479:         std::cout << "|x|---------->" << std::endl;
480:       }
481:     }
482:     std::cout << std::endl;
483:   }
484: }
485:
486: // N semi-independent (soft) Pomeron exchanges, 4-momentum conserved
487: //
488: // ------------------------------------> remnant 1
489: //       |       |       |       |
490: //      -- s0    |      -- s2     -- s(N-1)
491: // s    |       |       |       |
492: //       |      -- s1    |  ...  |
493: //       |       |       |       |
494: // ------------------------------------> remnant 2
495: //
496: //
497: // 1. Note that this problem is indeed quite sensitive to the (ad-hoc)
498: // beam-fragment
```



```
499: // treatment -> visible for example in multiplicity (double-NBD like)
500: // spectrum peak and shape etc. Here, we do not try to handle any color dof, but
501: // just
502: // put some boundary conditions and simple Regge-like x-dependence.
503: //
504: // 2. The energy evolution of the multiplicity spectrum density dN/deta is
505: // very sensitive to the Poisson-ansatz-eikonal process, naturally. Often seen
506: // easy powerlaw
507: // parametrization is not easy to obtain simultaneously with other observables.
508: // This just reminds that a very wide simultaneous set of observables is always
509: // needed
510: // <=> Rivet analysis.
511: //
512: std::complex<double> MQuasiElastic::PolySoft(const std::vector<double> &randvec) {
513:   // Boundary conditions
514:   const double XMIN = 1e-6;
515:   const double XMAX = 1.0;
516:   // const double XSTEP = 1e-5;
517:   const double MMIN      = 3.0;
518:   const double REM_M2_MIN = pow2(10.0);
519:   const double DELTA     = 0.10;
520:
521:   int         outertrials = 0;
522:   const int MAXTRIAL    = 1e4;
523:
524:   // Get the random number of inelastic cut Pomerons
525:   unsigned int N = 0;
526:
527:   if (!Eikonal.IsInitialized()) {
528:     throw std::invalid_argument("MQuasiElastic::PolySoft: Set POMLOOP = true");
529:   }
530:   Eikonal.S3GetRandomCutsBt(N, bt, random.rng);
531:
532:   // N = Eikonal.S3GetRandomCuts(random.rng);
533:   // printf("N = %d, bt = %0.3f \n", N, bt);
534:
535:   // ------------------------------------------------------
536:   // etree.resize(N);
537:   etree.resize(N + 2);
538:
539:   // Beams in
540:   etree[0].p1i = lts.pbeam1;  // ######
541:   etree[0].p2i = lts.pbeam2;  // ######
542:
543:   // Dirichlet parameters
544:   const double        DIRALPHA = 0.85;
545:   std::vector<double> alphavec(N, DIRALPHA);  // "Uniform case"
546:
547:   // Random integer from [1,NBins-1]
548:   // const double Q2 = 1.0;
549:
550:   /*
551:   // Evaluate Proton Structure Function
552:   static std::vector<double> F2val(1/XSTEP, 0.0);
553:   static double MAXVAL = 0;
554:   for (const auto& n : indices(F2val)) {
555:
556:                   const double xval = XMIN + n*XSTEP;
557:                   F2val[n] = gra::form::F2xQ2(xval, Q2);
558:                   MAXVAL = (F2val[n] > MAXVAL) ? F2val[n] : MAXVAL;
559:   }
560:   std::discrete_distribution<int> DIST(F2val.begin(), F2val.end());
561:   */
562:
563:   while (true) {
564:     bool faulty = false;
565:     for (std::size_t i = 0; i < N; ++i) {
566:       const double s = (etree[i].p1i + etree[i].p2i).M2();
567:
568:       // Draw Bjorken-x
569:       double x1 = 0;
570:       double x2 = 0;
571:
572:       const double p1_m2 = etree[i].p1i.M2();
573:       const double p2_m2 = etree[i].p2i.M2();
574:
575:       // No Q^2 dependence taken into account here
576:       // We assume xG(x) ~ 1/x^(DELTA) <=> G(x) = 1/x^(1+DELTA)
577:       x1 = random.PowerRandom(XMIN, XMAX, -(DELTA));
578:       x2 = random.PowerRandom(XMIN, XMAX, -(DELTA));
579:
580:       /*
581:       // Structure function parametrization
```



```
582:
583:       //x1 = XMIN + DIST(rng) * XSTEP;
584:       //x2 = XMIN + DIST(rng) * XSTEP;
585:
586:       // Acceptance-Rejection
587:       while (true) {
588:               const int n = RANDI(rng);
589:               if (random.U(0,1) < F2val[n] / MAXVAL) {
590:                       x1 = XMIN + n * XSTEP;
591:                       break;
592:               }
593:       }
594:       while (true) {
595:               const int n = RANDI(rng);
596:               if (random.U(0,1) < F2val[n] / MAXVAL) {
597:                       x2 = XMIN + n * XSTEP;
598:                       break;
599:               }
600:       }
601:       */
602:
603:       // Pick gaussian Fermi pt of Pomerons
604:       const double sigma = 0.4;  // GeV
605:       const double pt1   = msqrt(pow2(random.G(0, sigma)) + pow2(random.G(0, sigma)));
606:       const double pt2   = msqrt(pow2(random.G(0, sigma)) + pow2(random.G(0, sigma)));
607:       const double phi1  = random.U(0, 2.0 * PI);
608:       const double phi2  = random.U(0, 2.0 * PI);
609:
610:       // Pick remnant mass^2
611:       double p1_m2 = p1_m2;
612:       double p4_m2 = p2_m2;
613:
614:       if (i == N - 1) {  // Excite remnants (one could excite also intermediate)
615:         // Max operator for low energies
616:         p3_m2 = random.PowerRandom(REM_M2_MIN, pow2(etree[i].p1i.E()), -(1 + DELTA));
617:         p4_m2 = random.PowerRandom(REM_M2_MIN, pow2(etree[i].p2i.E()), -(1 + DELTA));
618:       }
619:
620:       // Construct remnant protons
621:       etree[i].p1f.Set(pt1 * std::cos(phi1), pt1 * std::sin(phi1),
622:                        (x1 - 1) * msqrt(-4 * p1_m2 + s) / 2,
623:                        msqrt(p3_m2 + pow2(pt1) + (-p1_m2 + s / 4) * pow2(x1 - 1)));
624:       etree[i].p2f.Set(pt2 * std::cos(phi2), pt2 * std::sin(phi2),
625:                        -(x2 - 1) * msqrt(-4 * p2_m2 + s) / 2,
626:                        msqrt(p4_m2 + pow2(pt2) + (-p2_m2 + s / 4) * pow2(x2 - 1)));
627:
628:       etree[i].q1 = etree[i].p1i - etree[i].p1f;
629:       etree[i].q2 = etree[i].p2i - etree[i].p2f;
630:       // etree[i].q1 = lts.pbeam1 - etree[i].p1f;
631:       // etree[i].q2 = lts.pbeam2 - etree[i].p2f;
632:
633:       etree[i].k = etree[i].q1 + etree[i].q2;
634:
635:       if (etree[i].k.M() < MMIN) {
636:         // printf("Faulty: M(i=%d/N=%d) = %0.1f [x1 = %0.4f, x2 = %0.4f]
637:         // \n", i, N, etree[i].k.M(), x1,x2);
638:         faulty = true;
639:         break;
640:       }
641:
642:       // ------------------------------------------------------------------
643:       // Evaluate matrix element (RESERVATION)
644:       // ...
645:       // ------------------------------------------------------------------
646:
647:       // Remnants out
648:       if (i < N - 1) {
649:         etree[i + 1].p1i = etree[i].p1f;  // ######
650:         etree[i + 1].p2i = etree[i].p2f;  // ######
651:       }
652:     }
653:
654:     // Take forward and backward remnants energy-momentum
655:     etree[N].k     = etree[N - 1].p1f;
656:     etree[N + 1].k = etree[N - 1].p2f;
657:
658:     // Check Energy-Momentum Conservation here explicitly !!!
659:     if (!faulty) {
660:       M4Vec p_sum;
661:       for (const auto &i : indices(etree)) {
662:         p_sum += etree[i].k;
663:         // printf("M[%i] = %0.3E \n", i, etree[i].k.M());
664:       }
```



```
665:       if (!gra::math::CheckEMC(p_sum - (lts.pbeam1 + lts.pbeam2))) {
666:         // printf("M = %d \n", N);
667:         p_sum.Print();
668:         faulty = true;
669:       }
670:     }
671:
672:     if (faulty) {
673:       ++outertrials;
674:       if (outertrials > MAXTRIAL) {
675:         // std::cout << "MQuasiElastic::Random: Failed!" << std::endl;
676:         return 0.0;
677:       }
678:       continue;  // try again whole chain
679:     } else {
680:       // std::cout << "MQuasiElastic::Random: Success!" << std::endl;
681:       break;
682:     }
683:   }
684:   // printf("MQuasiElastic::PolySoft: OK with outertrials = %d \n",
685:   // outertrials);
686:
687:   return 1.0;
688: }
689:
690: // 3-dimensional phase space vector initialization
691: // The adaptive phase space boundaries here are taken into account by
692: // B3IntegralVolume() function.
693: bool MQuasiElastic::B3RandomKin(const std::vector<double> &randvec) {
694:   // Elastic case: s1 = s3, s2 = s4
695:   double s3 = lts.pbeam1.M2();
696:   double s4 = lts.pbeam2.M2();
697:
698:   // Set Diffractive mass boundaries
699:   // neutron + piplus + safe margin
700:   M2_f_min = std::max(gcuts.X1_min * lts.s, gra::math::pow2(1.3));
701:   M2_f_max = gcuts.X1_max * lts.s;
702:
703:   log_M2_f_min = std::log(M2_f_min);
704:   log_M2_f_max = std::log(M2_f_max);
705:
706:   lts.excite1 = false;
707:   lts.excite2 = false;
708:
709:   // Sample diffractive system masses
710:   if (ProcPtr.CHANNEL == "SD") {
711:     // Log-change of variable
712:     const double u = log_M2_f_min + (log_M2_f_max - log_M2_f_min) * randvec[1];
713:     const double r = std::exp(u);
714:
715:     // Choose random permutation
716:     if (random.U(0, 1) < 0.5) {
717:       s3         = r;
718:       lts.excite1 = true;
719:       lts.excite2 = false;
720:     } else {
721:       s4         = r;
722:       lts.excite1 = false;
723:       lts.excite2 = true;
724:     }
725:   } else if (ProcPtr.CHANNEL == "DD") {
726:     // Log-change of variable
727:     const double u1 = log_M2_f_min + (log_M2_f_max - log_M2_f_min) * randvec[1];
728:     const double r1 = std::exp(u1);
729:
730:     DD_M2_max     = gcuts.X1_max * (pow2(mp) * lts.s) / r1;
731:     log_DD_M2_max = std::log(DD_M2_max);
732:
733:     // Log-change of variable
734:     const double u2 = log_M2_f_min + (log_DD_M2_max - log_M2_f_min) * randvec[2];
735:     const double r2 = std::exp(u2);
736:
737:     // Choose random permutations
738:     if (random.U(0, 1) < 0.5) {
739:       s3 = r1;
740:       s4 = r2;
741:     } else {
742:       s4 = r1;
743:       s3 = r2;
744:     }
745:     lts.excite1 = true;
746:     lts.excite2 = true;
747:   }
```

```
748:
749:   // Calculate kinematically valid t-range
750:   const double s1 = lts.pbeam1.M2();
751:   const double s2 = lts.pbeam2.M2();
752:
753:   // Mandelstam t-range calculation
754:   gra::kinematics::Two2TwoLimit(lts.s, s1, s2, s3, s4, t_min, t_max);
755:
756:   // Then limit the "diffraction cone" due to screening loop limit
757:   t_min = std::min(std::max(-pow2(Eikonal.Numerics.MaxLoopKT), t_min), t_max);
758:
759:   // Logarithmic change of variable sampling
760:   const double A = std::abs(t_max);
761:   const double B = std::abs(t_min);
762:
763:   const double r =
764:       std::log(A + ZERO_EPS) + (std::log(B + ZERO_EPS) - std::log(A + ZERO_EPS)) * randvec[0];
765:   const double t = -std::exp(r);
766:
767:   return B3BuildKin(s3, s4, t);
768: }
769:
770: // Build kinematics for elastic, single and double diffractive 2->2 quasielastic
771: bool MQuasiElastic::B3BuildKin(double s3, double s4, double t) {
772:   static const double s1      = lts.pbeam1.M2();
773:   static const double s2      = lts.pbeam2.M2();
774:   static const M4Vec  beamsum = lts.pbeam1 + lts.pbeam2;
775:
776:   // Scattering angle based on invariants
777:   double theta = std::acos(kinematics::CosthetaStar(lts.s, t, s1, s2, s3, s4));
778:
779:   // Forward/backward solution flip (skip these, rare)
780:   if (std::cos(theta) < 0) { return false; }
781:   // theta = (std::cos(theta) < 0) ? gra::math::PI - theta : theta;
782:
783:   // Outgoing 4-momentum by Kallen (triangle) function in the center-of-momentum
784:   // frame
785:   const double pnorm = kinematics::DecayMomentum(lts.sqrt_s, msqrt(s3), msqrt(s4));
786:   M4Vec        p3(0, 0, pnorm, 0.5 * (lts.s + s3 - s4) / lts.sqrt_s);
787:   M4Vec        p4(0, 0, -pnorm, 0.5 * (lts.s + s4 - s3) / lts.sqrt_s);
788:
789:   // Transverse momentum by orienting with random rotation (theta,phi)
790:   const double phi = random.U(0.0, 2.0 * gra::math::PI);  // Flat phi
791:   gra::kinematics::Rotate(p3, theta, phi);
792:   gra::kinematics::Rotate(p4, theta, phi);
793:
794:   // ------------------------------------------------------------------
795:   // Now boost if asymmetric beams
796:   if (std::abs(beamsum.Pz()) > 1e-6) {
797:     constexpr int sign = 1;  // positive -> boost to the lab
798:     kinematics::LorentzBoost(beamsum, lts.sqrt_s, p3, sign);
799:     kinematics::LorentzBoost(beamsum, lts.sqrt_s, p4, sign);
800:   }
801:   // ------------------------------------------------------------------
802:
803:   lts.pfinal[1] = p3;
804:   lts.pfinal[2] = p4;
805:
806:   // Check Energy-Momentum conservation
807:   if (!gra::math::CheckEMC(beamsum - (lts.pfinal[1] + lts.pfinal[2]))) { return false; }
808:
809:   return B3GetLorentzScalars();
810: }
811:
812: // Build and check scalars
813: bool MQuasiElastic::B3GetLorentzScalars() {
814:   // Calculate Lorentz scalars
815:   lts.ss[1][1] = lts.pfinal[1].M2();
816:   lts.ss[2][2] = lts.pfinal[2].M2();
817:
818:   lts.t = (lts.pbeam1 - lts.pfinal[1]).M2();
819:   lts.u = (lts.pbeam1 - lts.pfinal[2]).M2();
820:
821:   // Test scalars
822:   if (lts.ss[1][1] > lts.s) return false;
823:   if (lts.ss[2][2] > lts.s) return false;
824:   if (lts.t > 0) return false;
825:   if (lts.u > 0) return false;
826:   /*
827:   printf("s = %E, s^1/2 = %E \n", lts.s, msqrt(lts.s));
828:   printf("t = %E, u = %E \n", lts.t, lts.u);
829:   printf("s3 = %E, s4 = %E \n", lts.ss[1][1], lts.ss[2][2]);
830:   printf("\n");
```



```
831:   */
832:   return true;
833: }
834:
835: // 1/2/3-Dim Integral Volume [t] x [M^2] x [M^2]
836: //
837: double MQuasiElastic::B3IntegralVolume() const {
838:   // Mandelstam t, log change of variable, volume times jacobian
839:   const double A     = std::abs(t_max);
840:   const double B     = std::abs(t_min);
841:   const double t_VOL = (std::log(B + ZERO_EPS) - std::log(A + ZERO_EPS)) * std::abs(lts.t);
842:
843:   if (ProcPtr.CHANNEL == "EL") {
844:     return t_VOL;
845:
846:   } else if (ProcPtr.CHANNEL == "SD") {
847:     // Jacobian from log-change of variable:
848:     // \int_a^b f(M2) dM2 = \int_{ln(a)}^{ln(b)} f(exp(u)) * exp(u) du, where u = ln(M2)
849:     const double J      = (lts.excitel) ? lts.ss[1][1] : lts.ss[2][2];
850:     const double M2_VOL = (log_M2_f_max - log_M2_f_min) * J;
851:     return t_VOL * M2_VOL;
852:
853:   } else if (ProcPtr.CHANNEL == "DD") {
854:     // Jacobian from log-change of variables
855:     const double J      = lts.ss[1][1] * lts.ss[2][2];
856:     const double M2_VOL = (log_M2_f_max - log_M2_f_min) * (log_DD_M2_max - log_M2_f_min) * J;
857:     return t_VOL * M2_VOL;
858:
859:   } else {
860:     return 0;
861:   }
862: }
863:
864: // Standard phase space for m1 + m2 -> m3 + m4
865: //
866: double MQuasiElastic::B3PhaseSpaceWeight() const {
867:   // expression -> 16 * pi * p^2 {if s >> m1,m2}
868:   const double norm = 16.0 * gra::math::PI *
869:                       pow2(lts.s * gra::kinematics::beta12(lts.s, lts.beam1.mass, lts.beam2.mass));
870:
871:   if (ProcPtr.CHANNEL == "EL") {
872:     return 1.0 / norm;
873:   } else if (ProcPtr.CHANNEL == "SD") {
874:     return 2.0 / norm;  // Factor of two in
875:                         // numerator from single
876:                         // diffraction left + right
877:   } else if (ProcPtr.CHANNEL == "DD") {
878:     return 1.0 / norm;
879:   } else if (ProcPtr.CHANNEL == "ND") {
880:     return 1.0;
881:   } else {
882:     return 0;
883:   }
884: }
885:
886: }  // namespace gra
887:
```



```
  1: // I/O aux functions
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: // C++
  7: #include <complex>
  8: #include <fstream>
  9: #include <iostream>
 10: #include <regex>
 11: #include <vector>
 12: //#include <experimental/filesystem>
 13:
 14: // C system functions
 15: #include <fcntl.h>
 16: #include <sys/resource.h>
 17: #include <sys/stat.h>
 18: #include <sys/statvfs.h>
 19: #include <sys/time.h>
 20: #include <sys/types.h>
 21: #include <sys/utsname.h>
 22: #include <time.h>
 23: #include <unistd.h>
 24:
 25: // Libraries
 26: #include "HepMC3/FourVector.h"
 27: #include "LHAPDF/LHAPDF.h"
 28:
 29: // Own
 30: #include "Granitti/M4Vec.h"
 31: #include "Granitti/MAux.h"
 32: #include "Granitti/MForm.h"
 33: #include "Granitti/MSpin.h"
 34: #include "Granitti/MSudakov.h"
 35:
 36: // Libraries
 37: #include "rang.hpp"
 38:
 39: namespace gra {
 40: namespace aux {
 41:
 42: // ------------------------------------------------------
 43: // FIXED HERE manually
 44:
 45: double      GetVersion() { return 1.051; }
 46: std::string GetVersionType() { return "release"; }
 47: std::string GetVersionDate() { return "26.02.2020"; }
 48: std::string GetVersionUpdate() { return "performance, IO, new HepMC3 & fixes"; }
 49:
 50: // ------------------------------------------------------
 51:
 52: // Check do we have terminal output (=true), or output to file (=false)
 53: static const bool IS_TERMINAL = isatty(fileno(stdout)) != 0;
 54:
 55: // Print input arguments
 56: void PrintArgv(int argc, char *argv[]) {
 57:   std::cout << rang::fg::green << "$ ";
 58:   for (int i = 0; i < argc; ++i) {
 59:     const std::string s = std::string(argv[i]);
 60:     if (s.find(' ') != std::string::npos) {  // e.g. "FP[CON]<F> -> pi+ pi-"
 61:       std::cout << "\"" << s << "\""
 62:                 << " ";
 63:     } else {
 64:       std::cout << s << " ";
 65:     }
 66:   }
 67:   std::cout << rang::fg::reset << std::endl;
 68: }
 69:
 70: // Download LHAPDFset automatically
 71: void AutoDownloadLHAPDF(const std::string pdfname) {
 72:   std::cout << rang::fg::red
 73:             << "aux::AutoDownloadLHAPDF: Trying automatic download:" << rang::fg::reset
 74:             << std::endl;
 75:   std::cout << std::endl;
 76:
 77:   // Get instal path and remove "\n"
 78:   std::string INSTALLPATH = aux::ExecCommand("lhapdf-config --prefix");
 79:   INSTALLPATH.erase(std::remove(INSTALLPATH.begin(), INSTALLPATH.end(), '\n'), INSTALLPATH.end());
 80:
 81:   if (INSTALLPATH.find("command not found") != std::string::npos) {
 82:     throw std::invalid_argument("aux::AutoDownloadLHAPDF: Failure: lhapdf-config command missing");
 83:   }
```



```
 84:
 85:   // Download, untar
 86:   std::string cmd1 = "wget http://lhapdfsets.web.cern.ch/lhapdfsets/current/" + pdfname +
 87:                      ".tar.gz -O- | " + " tar xz -C " + INSTALLPATH + "/share/LHAPDF";
 88:   std::cout << cmd1 << std::endl;
 89:   std::string OUTPUT1 = aux::ExecCommand(cmd1);
 90: }
 91:
 92: // Run terminal command, get output to std::string
 93: std::string ExecCommand(const std::string &cmd) {
 94:   std::array<char, 128>                 buffer;
 95:   std::string                           result;
 96:   std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(cmd.c_str(), "r"), pclose);
 97:   if (!pipe) { throw std::runtime_error("aux::ExecCommand:: popen() failed!"); }
 98:   while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) { result += buffer.data(); }
 99:   return result;
100: }
101:
102: std::string GetExecutablePath() {
103:   char     buff[2048];
104:   ssize_t len = ::readlink("/proc/self/exe", buff, sizeof(buff) - 1);
105:   if (len != -1) {
106:     buff[len] = '\0';
107:     return std::string(buff);
108:   }
109:   return "";
110: }
111:
112: // level 0 returns same as GetExecutablePath:
113: // ~ /home/user/granitti/bin/gr
114: //
115: // level 1 returns
116: // ~ /home/user/granitti/bin
117: //
118: // etc..
119: //
120: std::string GetBasePath(std::size_t level) {
121:   std::string s   = GetExecutablePath();
122:   char        sep = '/';
123:
124:   for (std::size_t k = 0; k < level; ++k) {
125:     // Search backwards from end of string
126:     size_t i = s.rfind(sep);
127:     if (i != std::string::npos) {
128:       s = s.substr(0, i);
129:     } else {
130:       return s;
131:     }
132:   }
133:   return s;
134: }
135:
136: /*
137: // alias
138: namespace fs = std::experimental::filesystem;
139:
140: std::string GetCurrentPath() {
141:                 fs::path cwd = std::experimental::filesystem::current_path();
142:                 return cwd.string();
143: }
144:
145:
146: // Folder exists
147: bool FileExist(const fs::path& p, fs::file_status s) {
148:         if (fs::status_known(s) ? fs::exists(s) : fs::exists(p)) {
149:                 return true;
150:         } else {
151:                 return false;
152:         }
153: }
154: */
155:
156: // Get filesize in bytes
157: std::uintmax_t GetFileSize(const std::string &filename) {
158:   /*
159:   namespace fs = std::experimental::filesystem;
160:   fs::path p = fs::current_path() / filename;
161:   return fs::file_size(p);
162:   */
163:
164:   struct stat stat_buf;
165:   int         rc = stat(filename.c_str(), &stat_buf);
166:   return rc == 0 ? stat_buf.st_size : 0;
```

```
167: }
168:
169: // Get Process Memory Usage (linux/BSD/OSX) in bytes
170: void GetProcessMemory(double &peak_use, double &resident_use) {
171:   // Peak memory
172:   struct rusage rusage;
173:   getrusage(RUSAGE_SELF, &rusage);
174:   peak_use = rusage.ru_maxrss * 1024L;
175:
176:   // Current memory
177:   long  rss = 0L;
178:   FILE *fp  = NULL;
179:   if ((fp = fopen("/proc/self/statm", "r")) == NULL) {
180:     resident_use = 0.0;
181:     return;
182:   }
183:   if (fscanf(fp, "%*s%ld", &rss) != 1) {  // Reading problem
184:     resident_use = 0.0;
185:   } else {  // fine
186:     resident_use = rss * (size_t)sysconf(_SC_PAGESIZE);
187:   }
188:   fclose(fp);
189: }
190:
191: // Get disk usage
192: void GetDiskUsage(const std::string &path, int64_t &size, int64_t &free, int64_t &used) {
193:   int64_t frsize;
194:   int64_t blocks;
195:   int64_t bfree;
196:
197:   struct statvfs buf;
198:   int            ret = statvfs(path.c_str(), &buf);
199:
200:   if (!ret) {
201:     frsize = buf.f_frsize;  // block size
202:     blocks = buf.f_blocks;  // blocks
203:     bfree  = buf.f_bfree;   // free blocks
204:
205:     size = frsize * blocks;
206:     free = frsize * bfree;
207:     used = size - free;
208:   }
209: }
210:
211: // Get available memory in bytes
212: unsigned long long TotalSystemMemory() {
213:   long pages     = sysconf(_SC_PHYS_PAGES);
214:   long page_size = sysconf(_SC_PAGE_SIZE);
215:   return pages * page_size;
216: }
217:
218: // System information
219: std::string SystemName() {
220:   struct utsname name;
221:   uname(&name);
222:
223:   std::string sysname(name.sysname);
224:   std::string nodename(name.nodename);
225:   std::string release(name.release);
226:   std::string version(name.version);
227:   std::string machine(name.machine);
228:
229:   std::string s = " ";
230:
231:   return sysname + s + release + s + version + s + machine;
232: }
233:
234: // Get system hostname
235: std::string HostName() {
236:   char hostname[2048];
237:   gethostname(hostname, 2048);
238:
239:   std::string str(hostname);
240:   return str;
241: }
242:
243: // Get current date/time, format is YYYY-MM-DD HH:mm:ss
244: // http://en.cppreference.com/w/cpp/chrono/c/strftime
245: // for more information about date/time format
246: const std::string DateTime() {
247:   time_t     now = time(0);
248:   struct tm tstruct;
249:   char      buf[80];
```



```
250:   tstruct = *localtime(&now);
251:
252:   strftime(buf, sizeof(buf), "%Y-%m-%d %X", &tstruct);
253:   return buf;
254: }
255:
256: // Print out progress bar visualization
257: void PrintProgress(double ratio) {
258:   if (ratio > 1.0) { ratio = 1.0; }
259:
260: #define BAR "||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||"
261:   const int WIDTH = 62;
262:   const int pos   = static_cast<int>(ratio * 100);
263:   const int left  = static_cast<int>(ratio * WIDTH);
264:   const int right = WIDTH - left;
265:
266:   if (gra::aux::IS_TERMINAL) {  // if no terminal, do not print!
267:     std::cout << rang::fg::green;
268:     printf("\r%3d%% [%.*s%*s]", pos, left, BAR, right, "");
269:     std::cout << rang::fg::reset;
270:     std::cout << std::flush;
271:   }
272: }
273:
274: // Clear the line, and move cursor to the left
275: void ClearProgress() {
276:   std::cout << "\33[2K"
277:             << "\r";
278: }
279:
280: // djb2 hash function (used for saving unique filenames)
281: unsigned long djb2hash(const std::string &s) {
282:   unsigned long hash = 5381;  // Magic number
283:   for (auto c : s) {
284:     hash = (hash << 5) + hash + c;  // same as: hash * 33 + c
285:   }
286:   return hash;
287: }
288:
289: // Read CSV file
290: void ReadCSV(const std::string &inputfile, std::vector<std::vector<std::string>> &output) {
291:   std::fstream file;
292:   file.open(inputfile);
293:   std::string line;
294:
295:   // Read every line from the stream
296:   while (getline(file, line)) {
297:     std::istringstream       stream(line);
298:     std::vector<std::string> columns;
299:     std::string              element;
300:
301:     // Every line element separated by separator
302:     while (getline(stream, element, ',')) { columns.push_back(element); }
303:     output.push_back(columns);
304:   }
305:   file.close();
306: }
307:
308: // Get JSON file input
309: std::string GetInputData(const std::string &inputfile) {
310:   // Check if exists
311:   if (gra::aux::FileExist(inputfile) == false) {
312:     std::string str = "gra::aux::GetInputData: Inputfile '" + inputfile + "' does not exist";
313:     throw std::invalid_argument(str);
314:   }
315:   // Create a JSON object from file
316:   std::ifstream ifs(inputfile);
317:   std::string   data((std::istreambuf_iterator<char>(ifs)), (std::istreambuf_iterator<char>()));
318:
319:   // https://json5.org/ features:
320:   // We use C++11 raw string literals here R"(text)"
321:   // so it is easier to type in expressions without problems with slashes
322:
323:   // JSON5: <Single and multi-line comments are allowed>
324:   // First this. Remove C style block comments /* */
325:   data = std::regex_replace(data, std::regex(R"(\/\*(?!\/)|[^*])*\*\/)"), "");
326:   // Second this. Remove C++ style single line comments "//"
327:   data = std::regex_replace(data, std::regex(R"([/]+.*)"), "");
328:
329:   // Remove "\n" or "\t" or "\r", needed for the following regex operations
330:   data = std::regex_replace(data, std::regex(R"(\n|\t|\r)"), "");
331:
332:   // JSON5: <Additional white space characters are allowed.>
```



```
333:   // Compress all extra spaces (compress to 1 space)
334:   // This also processes through the strings "", not problems for use,
335:   // but (IMPROVE THIS!)
336:   data = std::regex_replace(data, std::regex(R"(\s{2,})"), "");
337:
338:   // JSON5: <Objects may have a single trailing comma.>
339:   // Replace ",}" or ", }" with "}"
340:   data = std::regex_replace(data, std::regex(R",(,|, )}"), "}");
341:
342:   // JSON5: <Arrays may have a single trailing comma.>
343:   // Replace ",]" or ", ]" with "]"
344:   data = std::regex_replace(data, std::regex(R",(,|, )]"), "]");
345:
346:   // std::cout << std::endl << data << std::endl << std::endl;
347:
348:   return data;
349: }
350:
351: // Check is it integer digits
352: bool IsIntegerDigits(const std::string &str) {
353:   return str.find_first_not_of("-0123456789") == std::string::npos;
354: }
355:
356: // Return particle parity as a string
357: std::string ParityToString(int value) {
358:   if (value > 0) {
359:     return "+";
360:   } else if (value == 0) {
361:     return "";
362:   } else {
363:     return "-";
364:   }
365: }
366:
367: // Return particle charge as a string
368: std::string Charge2XtoString(int q3) {
369:   const std::string sign  = (q3 < 0) ? "-" : " ";
370:   const int         absq3 = std::abs(q3);
371:
372:   if (absq3 == 6)
373:     return sign + "2";
374:   else if (absq3 == 3)
375:     return sign + "1";
376:   else if (absq3 == 2)
377:     return sign + "2/3";
378:   else if (absq3 == 1)
379:     return sign + "1/3";
380:   else
381:     return "0";
382: }
383:
384: // Return spin as a string
385: std::string Spin2XtoString(int J2) {
386:   if (J2 == 10)
387:     return "5";
388:   else if (J2 == 9)
389:     return "9/2";
390:   else if (J2 == 8)
391:     return "4";
392:   else if (J2 == 7)
393:     return "7/2";
394:   else if (J2 == 6)
395:     return "3";
396:   else if (J2 == 5)
397:     return "5/2";
398:   else if (J2 == 4)
399:     return "2";
400:   else if (J2 == 3)
401:     return "3/2";
402:   else if (J2 == 2)
403:     return "1";
404:   else if (J2 == 1)
405:     return "1/2";
406:   else
407:     return "0";
408: }
409:
410: // Split a string to strings separated by delimiter
411: std::vector<std::string> SplitStr2Str(std::string input, const char delim, bool trimextraspace) {
412:   std::vector<std::string> output;
413:   std::stringstream        ss(input);
414:
415:   // String by string
```



```
416:   while (ss.good()) {
417:     std::string substr;
418:     std::getline(ss, substr, delim);
419:
420:     if (trimextraspace) { TrimExtraSpace(substr); }
421:     output.push_back(substr);
422:   }
423:   return output;
424: }
425:
426: // Split string to ints
427: std::vector<int> SplitStr2Int(std::string input, const char delim) {
428:   std::vector<int> output;
429:   std::stringstream ss(input);
430:
431:   // Get inputfiles by comma
432:   while (ss.good()) {
433:     std::string substr;
434:     std::getline(ss, substr, delim);
435:
436:     TrimExtraSpace(substr);
437:     output.push_back(std::stoi(substr));
438:   }
439:   return output;
440: }
441:
442: // Trim leading, extra and trailing spaces
443: void TrimExtraSpace(std::string &value) {
444:   value = std::regex_replace(value, std::regex(R"(^ +| +$|( ) +)"), "$1");
445: }
446:
447: void TrimLeadSpace(std::string &value) {
448:   value = std::regex_replace(value, std::regex(R"(^ +)"), "$1");
449: }
450:
451: void TrimTrailSpace(std::string &value) {
452:   value = std::regex_replace(value, std::regex(R" +$)"), "$1");
453: }
454:
455: void TrimEmptySpace(std::string &value) {
456:   value = std::regex_replace(value, std::regex(R"( +)"), "$1");
457: }
458:
459: void TrimAllSpace(std::string &value) {
460:   value = std::regex_replace(value, std::regex(R"([^\S\r\n]+)"), "$1");
461: }
462:
463:
464: // Extract words from a string
465: std::vector<std::string> Extract(const std::string &str) {
466:   std::vector<std::string> words;
467:   std::stringstream        ss(str);
468:   std::string              buff;
469:
470:   while (ss >> buff) { words.push_back(buff); }
471:   return words;
472: }
473:
474: // Check if file exists
475: bool FileExist(const std::string &name) {
476:   struct stat buffer;
477:   return (stat(name.c_str(), &buffer) == 0);
478: }
479:
480: void PrintNotice() {
481:   std::cout << rang::fg::red << "<NOTICE>\n\n"
482:             << "ZZZ    ZZ  ZZZZZZ  ZZZZZZZZ ZZ ZZZZZZ ZZZZZZZ  \n"
483:                "ZZZZ   ZZ ZZ    ZZ    ZZ    ZZ ZZ      ZZ     \n"
484:                "ZZ ZZ  ZZ ZZ    ZZ    ZZ    ZZ ZZ      ZZ     \n"
485:                "ZZ  ZZ ZZ ZZ    ZZ    ZZ    ZZ ZZ      ZZ     \n"
486:                "ZZ   ZZZZ  ZZZZZZ      ZZ    ZZ ZZZZZZ ZZZZZZZ  \n"
487:             << rang::fg::reset << std::endl;
488: }
489:
490: void PrintWarning() {
491:   std::cout << rang::fg::red << "<WARNING>\n\n"
492:             << "ZZ     ZZ  ZZZZZ  ZZZZZZ  ZZZ    ZZ ZZ ZZZ    ZZ  ZZZZZZ   \n"
493:                "ZZ     ZZ ZZ   ZZ ZZ  ZZ  ZZ ZZZZ   ZZ ZZ ZZZZ   ZZ ZZ     \n"
494:                "ZZ  Z  ZZ ZZZZZZZ ZZZZZZ  ZZ ZZ  ZZ ZZ ZZ ZZ ZZ  ZZZ  \n"
495:                "ZZ ZZZ ZZ ZZ   ZZ ZZ  ZZ ZZ ZZ  ZZ ZZ ZZ ZZ  ZZ ZZ    ZZ  \n"
496:                " ZZZ ZZZ  ZZ   ZZ ZZ  ZZ ZZ  ZZZZ ZZ ZZ  ZZZZ  ZZZZZZ   \n"
497:             << rang::fg::reset << std::endl;
498: }
```

```cpp

void PrintGameOver() {
  std::cout << "<GAME OVER>\n\n"
            << " ZZZZZZ   ZZZZZ  ZZZ    ZZZ ZZZZZZZ    ZZZZZZ  ZZ    ZZ ZZZZZZZ ZZZZZZ   \n"
               "ZZ       ZZ   ZZ ZZZZ  ZZZZ ZZ        ZZ    ZZ ZZ    ZZ ZZ      ZZ   ZZ  \n"
               "ZZ   ZZZ ZZZZZZZ ZZ ZZZZ ZZ ZZZZZ     ZZ    ZZ ZZ    ZZ ZZZZZ   ZZZZZZ   \n"
               "ZZ    ZZ ZZ   ZZ ZZ  ZZ  ZZ ZZ        ZZ    ZZ  ZZ  ZZ  ZZ      ZZ   ZZ  \n"
               " ZZZZZZ  ZZ   ZZ ZZ      ZZ ZZZZZZZ    ZZZZZZ    ZZZZ   ZZZZZZZ ZZ   ZZ  \n"
            << std::endl;
}

// Check updates online
void CheckUpdate() {
  const std::string tmpfile = "/tmp/GRANIITTI_VERSION_" + std::to_string(time(0)) + ".json";
  const std::string cmd =
      "curl -s -o " + tmpfile +
      " https://raw.githubusercontent.com/mieskolainen/GRANIITTI/master/VERSION.json";

  // Execute curl
  const int ret = system(cmd.c_str());

  if (ret != -1) {  // Success

    // Read and parse
    std::string    data;
    nlohmann::json j;

    try {
      data = gra::aux::GetInputData(tmpfile);
      j    = nlohmann::json::parse(data);
    } catch (...) {
      std::cout << std::endl;
      PrintBar("-", 80);
      std::cout << "Check your internet access -- could not update version information"
                << std::endl;
      PrintBar("-", 80);
      std::cout << std::endl;
    }

    try {
      const double      online_version = j.at("version");
      const std::string online_date    = j.at("date");

      if (GetVersion() < online_version) {  // Older version than online
        std::cout << std::endl;
        PrintBar("-", 80);
        std::cout << rang::style::bold << rang::fg::green << "New version " << online_version
                  << " (" << online_date << ") available at <github.com/mieskolainen/GRANIITTI>"
                  << rang::fg::reset << rang::style::reset << std::endl;
        std::cout << std::endl;
        std::cout << "To update, copy-and-run: " << std::endl;
        std::cout
            << "git pull origin master && source ./install/setenv.sh && make superclean && make -j4"
            << std::endl
            << std::endl;
        std::cout
            << "If compilation fails, check that HepMC3 and LHAPDF6 are as in requirements.txt"
            << std::endl;
        std::cout << "You can re-install them with cd install && source autoinstall.sh"
                  << std::endl;
        PrintBar("-", 80);
        std::cout << std::endl;
      } else if (std::abs(GetVersion() - online_version) < 1e-6) {  // Same as online
        std::cout << std::endl;
        PrintBar("-", 80);
        std::cout << rang::style::bold << "This version " << GetVersion() << " ("
                  << GetVersionDate() << ") is up to date with the online version"
                  << rang::style::reset << std::endl;
        PrintBar("-", 80);
        std::cout << std::endl;
      } else {  // This version is newer than oline
        std::cout << std::endl;
        PrintBar("-", 80);
        std::cout << rang::style::bold << rang::fg::green << "This version " << GetVersion() << " ("
                  << GetVersionDate() << ") is newer than the online version " << online_version
                  << " (" << online_date << ")" << rang::style::reset << std::endl;
        PrintBar("-", 80);
        std::cout << std::endl;
      }

      // Remove tmp file
      const std::string rmcmd = "rm " + tmpfile;
      if (system(rmcmd.c_str()) != 1) {
```



```cpp
        // success
      }

    } catch (...) {
      // do nothing
    }
  }

  // Create this version JSON
  CreateVersionJSON();
}

void CreateVersionJSON() {
  // Get program path and output this VERSION.json
  const std::string output = GetBasePath(2) + "/VERSION.json";
  const std::string MSG =
      "{\n \"name\": \"%s\",\n \"version\": %0.3f,\n \"type\": \"%s\",\n \"date\": \"%s\",\n "
      "\"update\": \"%s\"\n}\n";

  FILE *file = fopen(output.c_str(), "w");
  fprintf(file, MSG.c_str(), "GRANIITTI", GetVersion(), GetVersionType().c_str(),
          GetVersionDate().c_str(), GetVersionUpdate().c_str());
  fclose(file);
}


std::string GetVersionString() {
  char buff[100];
  snprintf(buff, sizeof(buff), "Version %0.3f (%s) %s", GetVersion(), GetVersionType().c_str(),
           GetVersionDate().c_str());
  std::string str = buff;
  return str;
}

std::string GetVersionTLatex() {
  char         buff[100];
  const double version = GetVersion();
  snprintf(buff, sizeof(buff), "#color[16]{#scale[0.6]{GRANIITTI #scale[0.8]{%0.3f}}}", version);
  std::string str = buff;
  return str;
}

std::string GetWebTLatex() {
  char buff[100];
  snprintf(buff, sizeof(buff), "#color[16]{#scale[0.5]{#LTgithub.com/mieskolainen#GT}}");
  std::string str = buff;
  return str;
}

void PrintFlashScreen(rang::fg pcolor) {
  std::cout << std::endl;
  gra::aux::PrintBar("-");
  std::cout << pcolor
            << ".`````````````````````     ````   ``                                   ``   ```   ``\n"
               ".```..````````````    ` ``         `                                        ``    ``\n"
               ".``..:``````````` ```.``````.`.```                                           `    ``\n"
               ".``.`.```````   ````     ``-.```.````                                       ``` ``\n"
               "..-.`-.``    `.`   `..``---/--.......``                            ````````````\n"
               "..:..-.`````.`   `.--oys+ohms:..-.`..```                         ``````````````\n"
               ".-:.-...`..      .ooshyddddddddh+::...``...               ````````````````\n"
               "..-......        `yhhdhhmhddddmdy-...-...`                      ````````````\n"
               ":::::::.`    `    `--/ohdhdhddyo-`   `..`..                                  `\n"
               "so/-.`  `.`.-````-...++/syhdho`        `..                                  `\n"
               "``..--.:o-..--..-/-::-.-:/ohh/            `                              `` `\n"
               "`.-::-.:yyh/`--.`s+ohso:/shhd:                                          ``  `-\n"
               ".:/-s+o+syhs--:/-..-oydyshhhho`                                   ````` ``./\n"
               ".-+:-+shhhys+ysyh//sshyy/:oohyo.```````` `````` ```````````...::..  `.-oh\n"
               "-..-..:/+shhhyhhhyhysso.   ``:ys:----...........------:::::++oo-  `.-:ohd\n"
               "--+:--//:::/+oyyshhs+`   ``   .+so+++++++++++/-://+++++++++o+/-```-/yhhys+\n"
               "`/dysyhs/:::-:o+/-.`   `...     `..::+oooooo+:-``.:+oooo+/:-.```.:/ssso/.``\n"
               "``:oossso-.--.-.     ..`...`..`.``...`-/ossys+ssoso+ys:--..::/:/--.-//:```.\n"
               "`````````  `   `-.` `....```...:++.`./o:+-:/-::/-----...--.`-::::+ys/.`.:/sy\n"
               "```...-...`/......  ` `..`...`yo+:/o.:oshssy/.  ``  `.--.--:+sys+-` -+ossss\n"
               "``.//-/++-sh+-.-..`-`.-.--.- :hhho/+//:/yyyhhs+::+o+/++ssssso:.`-::oyssyyy\n"
               "`/yhoshhhh////++:.....`:yy+/...sddhs/:////:-/++ooo+oyssyoo:.`:/+oosssshddd\n"
               "`dddyhydhdyyddhddo``.+.`+/oyoh-..+yddho/-......--:----://+osyyhho++/--::/+\n"
               "/hddhmhmhmdmddhdds:+s+:+/-:++hhh//.-/++osssso++/-.````.-://::-.....`......\n"
            << std::endl << std::endl;
  std::cout << rang::fg::reset;
}

void PrintVersion() {
  std::cout << GetVersionString() << std::endl;
```



```cpp
  std::cout << rang::style::bold << "<github.com/mieskolainen/GRANIITTI>" << rang::style::reset
            << std::endl
            << std::endl;
  std::cout << "References: arXiv:1910.06300 [hep-ph]" << std::endl;
  std::cout << std::endl;
  std::cout << "(c) 2017-2020 Mikael Mieskolainen" << std::endl;
  std::cout << "<m.mieskolainen@imperial.ac.uk>" << std::endl;
  std::cout << std::endl;
  std::cout << "<opensource.org/licenses/GPL-3.0>" << std::endl;
  std::cout << "<opensource.org/licenses/MIT>" << std::endl;
}

// Print horizontal bar
void PrintBar(std::string str, unsigned int N) {
  for (std::size_t k = 0; k < N; ++k) { std::cout << str; }
  std::cout << std::endl;
}

// Create output directory if it does not exist
void CreateDirectory(std::string fullpath) {
  struct stat st = {0};
  if (stat(fullpath.c_str(), &st) == -1) { mkdir(fullpath.c_str(), 0700); }
}

// Get commandline arguments which are split by @... @... tagging syntax
std::vector<OneCMD> SplitCommands(const std::string &fullstr) {
  // Find @ blocks [ ... ]
  std::vector<std::size_t> pos = FindOccurance(fullstr, "@");
  std::vector<std::string> subcmd;

  if (pos.size() != 0) {
    // Take full string from the first @ till end
    std::string str = fullstr.substr(pos[0]);

    // Split multiple @id{} @id{} ... @id{} occurances
    if (pos.size() >= 2) {
      for (std::size_t i = 0; i < pos.size(); ++i) {
        const int start = pos[i] + 1;  // +1 so we skip @
        const int end   = (i < pos.size() - 1) ? pos[i + 1] : str.size();
        subcmd.push_back(str.substr(start, end - start));
      }
    } else {
      subcmd.push_back(str.substr(1));  // +1 so we skip @
    }
  }

  // Loop over @ blocks
  std::vector<OneCMD> cmd;

  for (std::size_t i = 0; i < subcmd.size(); ++i) {
    // ** Remove whitespace **
    TrimExtraSpace(subcmd[i]);

    std::string                        id;
    std::vector<std::string>           target;
    std::map<std::string, std::string> arg;

    // Check we find [] brackets
    std::size_t left  = subcmd[i].find("[");
    std::size_t right = subcmd[i].find("]");

    // Incomplete brackets
    if (left == std::string::npos && right != std::string::npos) {
      throw std::invalid_argument("SplitCommands: Incomplete @R[]{} syntax with missing [");
    }
    // Incomplete brackets
    else if (left != std::string::npos && right == std::string::npos) {
      throw std::invalid_argument("SplitCommands: Incomplete @R[]{} syntax with missing ]");
    }
    // Found [] brackets
    else if (left != std::string::npos && right != std::string::npos) {
      // Read (multiple) targets out
      const std::string TARGET_str = subcmd[i].substr(left + 1, right - left - 1);
      if (TARGET_str != "") {  // do not process empty
        target = SplitStr2Str(TARGET_str, ',');
      }

      // Remove [...] block including brackets
      subcmd[i].erase(left, right - left + 1);
    }

    // Check we find {} brackets
    std::size_t Lpos = subcmd[i].find("{", 0);
```



```cpp
    std::size_t Rpos = subcmd[i].find("}", 0);

    // Incomplete brackets
    if (Lpos == std::string::npos && Rpos != std::string::npos) {
      throw std::invalid_argument("SplitCommands: Incomplete @R[]{} syntax with missing {");
    }
    // Incomplete brackets
    else if (Lpos != std::string::npos && Rpos == std::string::npos) {
      throw std::invalid_argument("SplitCommands: Incomplete @R[]{} syntax with missing }");
    }

    // Singlet command @ID:VALUE
    if (Lpos == std::string::npos && Rpos == std::string::npos) {
      std::vector<std::string> strip = SplitStr2Str(subcmd[i], ':');

      // ID
      id = strip[0];

      if (strip.size() == 1) {
        arg["_SINGLET_"] = "true";  // default true, no : given
      } else if (strip.size() == 2) {
        arg["_SINGLET_"] = strip[1];
      } else {
        throw std::invalid_argument("gra::SplitCommands: @Syntax invalid with '" + subcmd[i] + "'");
      }

      // Block command @blaa{key:val,key:val,...}
    } else {  // We have {} block

      // ID
      id = subcmd[i].substr(0, Lpos);

      // Content {}

      // Now split all arguments inside {} by comma ','
      std::vector<std::string> keyvals =
          SplitStr2Str(subcmd[i].substr(Lpos + 1, Rpos - Lpos - 1), ',');

      // Add all
      for (std::size_t i = 0; i < keyvals.size(); ++i) {
        // Split using syntax definition: key:val
        std::vector<std::string> strip = SplitStr2Str(keyvals[i], ':');
        if (strip.size() != 2) {
          throw std::invalid_argument("gra::SplitCommands: @Syntax not good with brackets {" +
                                      keyvals[i] + "}");
        }

        // Strip spaces from the key
        TrimExtraSpace(strip[0]);

        arg[strip[0]] = strip[1];  // add to map
      }
    }

    // Add this command block
    OneCMD o;
    o.id     = id;
    o.target = target;
    o.arg    = arg;
    o.Print();  // For debug
    cmd.push_back(o);
  }
  return cmd;
}

// Find string occurances
std::vector<std::size_t> FindOccurance(const std::string &str, const std::string &sub) {
  // Holds all the positions that sub occurs within str
  std::vector<size_t> positions;
  std::size_t         pos = str.find(sub, 0);

  while (pos != std::string::npos) {
    positions.push_back(pos);
    pos = str.find(sub, pos + 1);
  }
  return positions;
}

}  // namespace aux
}  // namespace gra

```

./src/MSubProc.cc 1/2

```cpp
// (Sub)-Processes and Amplitude containers
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

// C++
#include <complex>
#include <random>
#include <vector>

// Own
#include "Granitti/MKinematics.h"
#include "Granitti/MSubProc.h"

// Libraries
#include "rang.hpp"

namespace gra {

// Add all processes
void MSubProc::CreateProcesses() {
  DeleteProcesses();

  pr.push_back(new PROC_0());
  pr.push_back(new PROC_1());
  pr.push_back(new PROC_2());
  pr.push_back(new PROC_3());
  pr.push_back(new PROC_4());
  pr.push_back(new PROC_5());
  pr.push_back(new PROC_6());
  pr.push_back(new PROC_7());
  pr.push_back(new PROC_8());
  pr.push_back(new PROC_9());
  pr.push_back(new PROC_10());
  pr.push_back(new PROC_11());
  pr.push_back(new PROC_12());
  pr.push_back(new PROC_13());
  pr.push_back(new PROC_14());
  pr.push_back(new PROC_15());
  pr.push_back(new PROC_16());
  pr.push_back(new PROC_17());
  pr.push_back(new PROC_18());
  pr.push_back(new PROC_19());
  pr.push_back(new PROC_20());
  pr.push_back(new PROC_21());
  pr.push_back(new PROC_22());
  pr.push_back(new PROC_23());
  pr.push_back(new PROC_24());
  pr.push_back(new PROC_25());
  pr.push_back(new PROC_26());
  pr.push_back(new PROC_27());
}

// Generic constructor, used for the first initialization
MSubProc::MSubProc(const std::vector<std::string>& istate, const std::string& mc) {
  for (const auto& i : aux::indices(istate)) { ConstructDescriptions(istate[i], mc); }
}

// Set spesific initial state and channel
void MSubProc::Initialize(const std::string& istate, const std::string& channel) {
  ISTATE  = istate;
  CHANNEL = channel;
  DeleteProcesses();
}

// Destructor
MSubProc::~MSubProc() { DeleteProcesses(); }

// Delete all processes
void MSubProc::DeleteProcesses() {
  for (const auto& i : aux::indices(pr)) { delete pr[i]; }
  pr.clear();  // Finally empty
}

void MSubProc::ConstructDescriptions(const std::string& istate, const std::string& mc) {
  CreateProcesses();

  // Construct labels
  for (const auto& i : aux::indices(pr)) {
    if (pr[i]->ISTATE == istate) {
      Processes.insert(std::make_pair(pr[i]->ISTATE + "[" + pr[i]->CHANNEL + "]<" + mc + ">",
                                      pr[i]->DESCRIPTION));
    }
```

./src/MSubProc.cc 2/2

```cpp
  }

  DeleteProcesses();  // Important!
}

// Activate spesific process
void MSubProc::ActivateProcess() {
  CreateProcesses();

  while (true) {
    for (const auto& i : aux::indices(pr)) {
      if (pr[i]->ISTATE == ISTATE && pr[i]->CHANNEL == CHANNEL) {
        // Fine
      } else {
        delete pr[i];
        pr.erase(pr.begin() + i);
        break;
      }
    }
    if (pr.size() <= 1) { break; }
  }
  if (pr.size() == 0) {
    throw std::invalid_argument("MSubProc::ActivateProcess: Unknown ISTATE = " + ISTATE +
                                " or CHANNEL = " + CHANNEL);
  }
}

// Print out process lists
std::vector<std::string> MSubProc::PrintProcesses() const {
  // Iterate through processes
  std::vector<std::string>                                                procstr;
  std::map<std::string, std::vector<std::string>>::const_iterator it = Processes.begin();

  while (it != Processes.end()) {
    procstr.push_back(it->first);

    if (it->second.size() == 0) {
      printf("%25s  =       ", it->first.c_str());
    } else if (it->second.size() == 1) {
      printf("%25s  =  %-43s", it->first.c_str(), it->second[0].c_str());
    } else if (it->second.size() == 2) {
      printf("%25s  =  %-43s  |  %-20s", it->first.c_str(), it->second[0].c_str(),
                                          it->second[1].c_str());
    } else if (it->second.size() >= 3) {
      printf("%25s  =  %-43s  |  %-20s  | %-10s", it->first.c_str(), it->second[0].c_str(),
                                                   it->second[1].c_str(), it->second[2].c_str());
    }
    std::cout << "\n";
    ++it;
  }
  return procstr;
}

// Check if process exists
bool MSubProc::ProcessExist(std::string str) const {
  if (Processes.find(str) != Processes.end()) { return true; }
  return false;
}

// Return process description string
std::vector<std::string> MSubProc::GetProcessDescriptor(std::string str) const {
  if (!ProcessExist(str)) {
    throw std::invalid_argument("MSubProc::GetProcessDescriptor: Process by name " + str +
                                " does not exist");
  }
  return Processes.find(str)->second;
}

// Wrapper function
double MSubProc::GetBareAmplitude2(gra::LORENTZSCALAR& lts) {
  if (pr.size() == 0) { ActivateProcess(); }
  return pr[0]->Amp2(lts);
}

}  // namespace gra
```

./src/MH1.cc 1/7

```cpp
// Templated 1D-histogram class with real or complex weights
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

// C++
#include <algorithm>
#include <complex>
#include <iostream>
#include <valarray>
#include <vector>

// Own
#include "Granitti/MH1.h"
#include "Granitti/MMath.h"

// Libraries
#include "rang.hpp"

namespace gra {
// Constructor
template <class T>
MH1<T>::MH1(int xbins, double xmin, double xmax, std::string namestr) {
  name = namestr;
  // Init
  ResetBounds(xbins, xmin, xmax);
  FILLBUFF = false;
}

// Constructor with only number of bins
template <class T>
MH1<T>::MH1(int xbins, std::string namestr) {
  name    = namestr;
  XBINS   = xbins;
  FILLBUFF = true;
}

// Empty constructor
template <class T>
MH1<T>::MH1() {
  XBINS    = 50;  // Default
  FILLBUFF = true;
}

// Destructor
template <class T>
MH1<T>::~MH1() {}


// Raw output
template <class T>
void MH1<T>::RawOutput() const {
  printf("#binlow,binhigh,value,error \n");

  for (std::size_t idx = 0; idx < static_cast<unsigned int>(XBINS); ++idx) {
    // This will take care of amplitude versus amplitude squared
    const double value     = GetPositiveDefinite(idx);
    const double value_err = GetBinError(idx);

    printf("%0.5E,%0.5E,%0.5E,%0.5E\n", GetBinXVal(idx, -1), GetBinXVal(idx, 1), value, value_err);
  }
}

template <class T>
void MH1<T>::Print(double width) const {
  if (!(fills > 0)) {  // No fills
    std::cout << "MH1::Print: <" << name << "> Fills = " << fills << std::endl;
    return;
  }

  // Histogram name
  std::cout << "MH1::Print: <" << name << ">" << std::endl;

  const std::size_t N         = XBINS * width;          // Number of ascii columns (PARAMETER)
  double            maxvisual = 1.1 * GetMaxWeight();  // Give some % space to top

  std::cout << "          |";  // Empty top left corner
  for (std::size_t i = 0; i < N; ++i) { std::cout << "="; }
  std::cout << "| " << std::endl;

  for (std::size_t idx = 0; idx < static_cast<unsigned int>(XBINS); ++idx) {
    const int boundary = 0;  // (lower,center,upper)
    printf("%9.2E |", GetBinXVal(idx, boundary));
```

./src/MH1.cc 2/7

```cpp

    // This will take care of amplitude versus amplitude squared
    const double value     = GetPositiveDefinite(idx);
    const double value_err = GetBinError(idx);

    // Visualization values scaled between [0,maxw)
    const double w = (maxvisual > 0) ? value / maxvisual : 0;

    // Y-AXIS index
    const std::size_t ind = std::round(w * (N - 1));

    const int Nc = GetBinCount(idx);
    if (Nc > 0) {  // non-zero bin

      // Error on value (only on double histograms)
      const double        relative_err = value_err / value;
      const std::size_t U              = std::ceil(w * (N - 1) * (1 + relative_err));   // up
      const std::size_t D              = std::floor(w * (N - 1) * (1 - relative_err));  // down
      // Print now
      for (std::size_t j = 0; j < N; ++j) {
        if (j == ind) {
          std::cout << rang::bg::magenta << "+" << rang::bg::reset;
        } else if (j >= D && j != ind && j <= U) {
          std::cout << rang::bg::magenta << " " << rang::bg::reset;
        } else {
          std::cout << " ";
        }
      }
      // No events in the bin
    } else {
      for (std::size_t j = 0; j < N; ++j) { std::cout << " "; }
    }
    const double BINWIDTH = (XMAX - XMIN) / XBINS;
    // Raw value +- error
    // printf("| %0.1E +- %0.1E \n", value, value_err);
    // raw value +- error | dO/dx (differential cross section, for example)
    printf("| %0.1E +- %0.1E [dO/dx] = %0.1E \n", value, value_err, value / fills / BINWIDTH);
  }

  std::cout << "          |";  // Empty bottom left corner
  for (std::size_t i = 0; i < N; ++i) { std::cout << "="; }
  std::cout << "| " << std::endl;

  // Print statistics
  std::cout << "<binned statistics>" << std::endl;
  std::pair<double, double> valerr = WeightMeanAndError();
  printf(" <W> = %0.3E +- %0.3E  [F = %lld | O/O = %lld/%lld]\n", valerr.first, valerr.second,
         fills, underflow, overflow);

  const double mean   = GetMean(1);
  const double sqmean = GetMean(2);
  printf(" <X> = %0.3f, <X^2> = %0.3f, <X^2> - <X>^2 = %0.3f\n", mean, sqmean,
         sqmean - std::pow(mean, 2));

  std::cout << std::endl;
}

template <class T>
std::pair<double, double> MH1<T>::WeightMeanAndError() const {
  const double N    = fills;  // Need to use number of total fills here, not counts in bins
  const double val  = std::abs(SumWeights()) / N;
  const double err2 = std::abs(SumWeights2()) / N - gra::math::pow2(val);
  const double err  = gra::math::msqrt(err2 / N);

  return {val, err};
}

// Get valarrays containing X values and corresponding density values
template <class T>
void MH1<T>::GetXPositiveDefinite(std::valarray<double> &x, std::valarray<double> &y) const {
  x.resize(XBINS);
  y.resize(XBINS);
  for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
    x[i] = GetBinXVal(i, 0);  // Get center value
    y[i] = GetPositiveDefinite(i);
  }
}

// Histogram mean (based on binned values)
template <class T>
double MH1<T>::GetMean(int power) const {
  double       sum  = 0.0;
  double       norm = 0.0;  // Normalization
```

```
167:   const double binwidth = (XMAX - XMIN) / XBINS;
168:
169:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
170:     const double value  = std::pow(binwidth * (i + 1) - binwidth / 2.0 + XMIN, power);
171:     const double weight = GetPositiveDefinite(i);
172:     sum += weight * value;
173:     norm += weight;
174:   }
175:   if (norm > 0) {
176:     return sum / norm;
177:   } else {
178:     return 0.0;
179:   }
180: }
181:
182: template <class T>
183: bool MH1<T>::ValidBin(int idx) const {
184:   if (idx >= 0 && idx < XBINS) { return true; }
185:   return false;
186: }
187:
188: // Unweighted fill
189: template <class T>
190: void MH1<T>::Fill(double xvalue) {
191:   // Call the weighted with weight 1.0
192:   Fill(xvalue, 1.0);
193: }
194:
195: // Weighted fill
196: template <class T>
197: void MH1<T>::Fill(double xvalue, T weight) {
198:   if (!FILLBUFF) {  // Normal filling
199:
200:     fills += 1;
201:
202:     // Find out bin
203:     const int idx = GetIdx(xvalue, XMIN, XMAX, XBINS, LOGX);
204:
205:     if (idx == -3) { nanflow += 1; }
206:     if (idx == -1) { underflow += 1; }
207:     if (idx == -2) { overflow += 1; }
208:
209:     if (ValidBin(idx)) {
210:       weights[idx] += weight;
211:       weights2[idx] += std::abs(std::conj(weight) * weight);
212:       counts[idx] += 1;
213:     }
214:   } else {  // Autorange initialization
215:     buff_values.push_back(xvalue);
216:     buff_weights.push_back(weight);
217:
218:     if (buff_values.size() > static_cast<unsigned int>(AUTOBUFFSIZE)) { FlushBuffer(); }
219:   }
220: }
221:
222: // Automatic histogram range algorithm
223: template <class T>
224: void MH1<T>::FlushBuffer() {
225:   if (FILLBUFF && buff_values.size() > 0) {
226:     FILLBUFF = false;  // no more filling buffer
227:
228:     // Find out mean
229:     double mu   = 0;
230:     double sumW = 0;
231:     for (std::size_t i = 0; i < buff_values.size(); ++i) {
232:       mu += std::abs(buff_values[i] * buff_weights[i]);
233:       sumW += std::abs(buff_weights[i]);
234:     }
235:     if (sumW > 0) { mu /= sumW; }
236:
237:     // Variance
238:     double var = 0;
239:     for (std::size_t i = 0; i < buff_values.size(); ++i) {
240:       var += std::abs(buff_weights[i]) * std::pow(buff_values[i] - mu, 2);
241:     }
242:     if (sumW > 0) { var /= sumW; }
243:
244:     // Minimum and maximum
245:     auto        it1    = std::min_element(buff_values.begin(), buff_values.end());
246:     auto        it2    = std::max_element(buff_values.begin(), buff_values.end());
247:     const double minval = *it1;
248:     const double maxval = *it2;
249:
```



```
250:     // Set new histogram bounds
251:     const double std = std::sqrt(std::abs(var));
252:
253:     double xmin = mu - 2.5 * std;
254:     double xmax = mu + 2.5 * std;
255:
256:     // A numerical failure may happen with variance calculation, then use this
257:     if (std::isnan(xmin) || std::isnan(xmax)) {
258:       xmin = minval;
259:       xmax = maxval;
260:     }
261:
262:     // If symmetric setup set by user
263:     if (AUTOSYMMETRY) {
264:       double val = (std::abs(xmin) + std::abs(xmax)) / 2.0;
265:       xmin       = -val;
266:       xmax       = val;
267:     }
268:
269:     // We have only positive values, such as invariant mass
270:     if (minval > 0.0) { xmin = std::max(0.0, xmin); }
271:
272:     ResetBounds(XBINS, xmin, xmax);
273:
274:     // Fill buffered events
275:     for (std::size_t i = 0; i < buff_values.size(); ++i) { Fill(buff_values[i], buff_weights[i]); }
276:
277:     // Clear buffers
278:     buff_values.clear();
279:     buff_weights.clear();
280:   }
281: }
282:
283: // Reset histogram completely
284: template <class T>
285: void MH1<T>::ResetBounds(int xbins) {
286:   ResetBounds(xbins, 0.0, 0.0);
287:   FILLBUFF = true;  // Autorange on, no explicit bounds
288: }
289:
290: // Reset histogram completely
291: template <class T>
292: void MH1<T>::ResetBounds(int xbins, double xmin, double xmax) {
293:   XMIN  = xmin;
294:   XMAX  = xmax;
295:   XBINS = xbins;
296:
297:   // Init
298:   std::vector<T> null(XBINS, 0.0);
299:   weights  = null;
300:   weights2 = null;
301:   counts   = std::vector<long long int>(XBINS, 0);
302:
303:   Clear();           // Call also this!
304:   FILLBUFF = false;  // No autorange, explicit bounds provided
305: }
306:
307: // Clear the histogram data but keep the bounds
308: template <class T>
309: void MH1<T>::Clear() {
310:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
311:     weights[i]  = 0.0;
312:     weights2[i] = 0.0;
313:     counts[i]   = 0;
314:   }
315:   fills    = 0;
316:   underflow = 0;
317:   overflow  = 0;
318:   nanflow   = 0;
319: }
320:
321: // Sum over all histogram bin weights
322: template <class T>
323: T MH1<T>::SumWeights() const {
324:   T sum = 0.0;
325:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) { sum += weights[i]; }
326:   return sum;
327: }
328:
329: // Sum over all histogram bin weights squared
330: template <class T>
331: T MH1<T>::SumWeights2() const {
332:   T sum = 0.0;
```



```
333:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) { sum += weights2[i]; }
334:   return sum;
335: }
336:
337: // Sum the number of bin counts (not the same as fills)
338: template <class T>
339: long long int MH1<T>::SumBinCounts() const {
340:   long long int sum = 0;
341:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) { sum += counts[i]; }
342:   return sum;
343: }
344:
345: // Get maximum histogram bin weight, for complex return |w|^2
346: template <class T>
347: double MH1<T>::GetMaxWeight() const {
348:   double maxval = 0;
349:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
350:     if (GetPositiveDefinite(i) > maxval) { maxval = GetPositiveDefinite(i); }
351:   }
352:   return maxval;
353: }
354:
355: // Get minimum histogram bin weight, for complex return |w|^2
356: template <class T>
357: double MH1<T>::GetMinWeight() const {
358:   double minval = 1e128;
359:   for (std::size_t i = 0; i < static_cast<unsigned int>(XBINS); ++i) {
360:     if (GetPositiveDefinite(i) < minval) { minval = GetPositiveDefinite(i); }
361:   }
362:   return minval;
363: }
364:
365: // Get number of event fills in the bin
366: template <class T>
367: long long int MH1<T>::GetBinCount(int idx) const {
368:   if (ValidBin(idx)) {
369:     return counts[idx];
370:   } else {
371:     return 0;
372:   }
373: }
374:
375: // Get weight of the bin, for complex return complex number
376: template <class T>
377: T MH1<T>::GetBinWeight(int idx) const {
378:   if (ValidBin(idx)) {
379:     return weights[idx];
380:   } else {
381:     return 0.0;
382:   }
383: }
384:
385: // Get weight of the bin, for complex return complex number
386: template <class T>
387: T MH1<T>::GetBinWeight2(int idx) const {
388:   if (ValidBin(idx)) {
389:     return weights2[idx];
390:   } else {
391:     return 0.0;
392:   }
393: }
394:
395: // Error estimate on bin
396: template <class T>
397: double MH1<T>::GetBinError(int idx) const {
398:   if (!ValidBin(idx)) { return 0.0; }
399:   double err = 0;
400:   if (GetBinCount(idx) > 0) {
401:     if (IsReal()) {
402:       err = std::sqrt(std::abs(GetBinWeight2(idx)));  // [\sum_i w_i^2)^(1/2)
403:     } else {
404:       err = std::abs(GetBinWeight2(idx));  // [\sum_i w_i^2)^(1/2)
405:     }
406:   }
407:   return err;
408: }
409:
410: // Return weight (double) or |w|^2 (complex case)
411: template <class T>
412: double MH1<T>::GetPositiveDefinite(int idx) const {
413:   if (ValidBin(idx)) {
414:     if (IsReal()) { return std::abs(weights[idx]); }
415:     return std::pow(std::abs(weights[idx]), 2);
```



```
416:   } else {
417:     return 0.0;
418:   }
419: }
420:
421: // Get bin index (idx) corresponding to value (xvalue)
422: template <class T>
423: void MH1<T>::GetBinIdx(double xvalue, int &idx) {
424:   // Find out bins
425:   idx = GetIdx(xvalue, XMIN, XMAX, XBINS, LOGX);
426: }
427:
428: // Get bin value in units of X for a given bin index
429: // boundary = -1,0,1 (lower, center, upper)
430: template <class T>
431: double MH1<T>::GetBinXVal(int idx, int boundary) const {
432:   if (idx > XBINS - 1) {
433:     throw std::invalid_argument("MH1::GetBinXVal: idx = " + std::to_string(idx) +
434:                                 " > XBINS-1 = " + std::to_string(XBINS - 1));
435:   }
436:   if (LOGX) {
437:     const double log10step = (std::log10(XMAX) - std::log10(XMIN)) / XBINS;
438:     if (boundary == -1) {
439:       return std::pow(10, std::log10(XMIN) + idx * log10step);
440:     } else if (boundary == 0) {
441:       return (std::pow(10, std::log10(XMIN) + idx * log10step) +
442:               std::pow(10, std::log10(XMIN) + (idx + 1) * log10step)) /
443:              2;
444:     } else if (boundary == 1) {
445:       return std::pow(10, std::log10(XMIN) + (idx + 1) * log10step);
446:     } else {
447:       throw std::invalid_argument("MH1::GetBinXVal: Bin boundary not valid (-1,0,1)");
448:     }
449:   } else {
450:     const double binwidth = (XMAX - XMIN) / XBINS;
451:     const double value    = XMIN + (idx + 1) * binwidth;
452:     if (boundary == -1) {
453:       return value - binwidth;
454:     } else if (boundary == 0) {
455:       return value - binwidth / 2.0;
456:     } else if (boundary == 1) {
457:       return value;
458:     } else {
459:       throw std::invalid_argument("MH1::GetBinXVal: Bin boundary not valid (-1,0,1)");
460:     }
461:   }
462: }
463:
464: // Get table/histogram index for linearly or base-10 logarithmically spaced
465: // bins
466: // Gives exact uniform filling within bin boundaries.
467: //
468: // In the logarithmic case, MINVAL and MAXVAL > 0, naturally.
469: //
470: //
471: // Underflow returns -1
472: // Overflow  returns -2
473: template <class T>
474: int MH1<T>::GetIdx(double value, double minval, double maxval, int nbins, bool logbins) const {
475:   if (std::isnan(value) || std::isinf(value)) { return -3; }
476:   if (value < minval) { return -1; }  // underflow
477:   if (value > maxval) { return -2; }  // overflow
478:
479:   int idx = 0;
480:   // Logarithmic binning
481:   if (logbins) {
482:     // Check do we have non-negative input
483:     idx = value > 0 ? std::floor(nbins * (std::log10(value) - std::log10(minval)) /
484:                                  (std::log10(maxval) - std::log10(minval)))
485:                     : -1;
486:     // Linear binning
487:   } else {
488:     const double BINWIDTH = (maxval - minval) / nbins;
489:     idx                   = std::floor((value - minval) / BINWIDTH);
490:   }
491:   return idx;
492: }
493:
494: // Instantiate (necessary for compilation)
495: template class MH1<double>;
496: template class MH1<std::complex<double>>;
497:
498: }  // namespace gra
```

```
499:
```



```
  1: // KISS fragmentation class
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: // C++
  7: #include <complex>
  8: #include <random>
  9: #include <valarray>
 10: #include <vector>
 11:
 12: // Own
 13: #include "Graniitti/M4Vec.h"
 14: #include "Graniitti/MAux.h"
 15: #include "Graniitti/MForm.h"
 16: #include "Graniitti/MFragment.h"
 17: #include "Graniitti/MKinematics.h"
 18: #include "Graniitti/MMath.h"
 19: #include "Graniitti/MMatrix.h"
 20: #include "Graniitti/MPDG.h"
 21: #include "Graniitti/MRandom.h"
 22:
 23: // Libraries
 24: #include "rang.hpp"
 25:
 26: using gra::aux::indices;
 27: using gra::math::msqrt;
 28: using gra::math::pow2;
 29:
 30: using gra::PDG::mp;
 31: using gra::PDG::mpi;
 32:
 33: using gra::PDG::PDG_gamma;
 34: using gra::PDG::PDG_n;
 35: using gra::PDG::PDG_p;
 36: using gra::PDG::PDG_pi0;
 37: using gra::PDG::PDG_pim;
 38: using gra::PDG::PDG_pip;
 39:
 40: namespace gra {
 41:
 42: // Return decay status [RESERVATION!]
 43: //
 44: //
 45: void MFragment::GetDecayStatus(const std::vector<int> &pdgcode, std::vector<bool> &isstable) {
 46:   isstable.resize(pdgcode.size(), true);
 47:   for (const auto &i : indices(isstable)) {
 48:     isstable[i] = (pdgcode[i] == 111 || pdgcode[i] == 311) ? false : true;
 49:   }
 50: }
 51:
 52: // Sample from Hagedorn type parametrization
 53: //
 54: // This function is useful to generate distribution with
 55: // exponential at low pt, powerlaw at high pt.
 56: // Does not generate jet like topologies,
 57: // so this can be used as a strict null model (H0) for soft like topologies.
 58: //
 59: // [slow function, performance should be upgraded]
 60: //
 61: // Parameters <=>
 62: //  p0 = T/(q-1)
 63: //   n = 1/(q-1)
 64: //
 65: void MFragment::ExpPowRND(double q, double T, double maxpt, const std::vector<double> &mass,
 66:                           std::vector<double> &x, MRandom &rng) {
 67:   const unsigned int MAXTRIAL = 1e4;  // Safety break
 68:   const unsigned int Nbins    = 1e4;
 69:   const double       ptstep   = maxpt / Nbins;
 70:
 71:   // Calculate pt-bin values
 72:   std::vector<double> ptval(Nbins);
 73:   for (std::size_t i = 0; i < Nbins; ++i) { ptval[i] = i * ptstep; }
 74:
 75:   // Random integer from [0,Nbins-1]
 76:   // C++11, thread_local is also static
 77:   thread_local std::uniform_int_distribution<int> RANDI(0, Nbins - 1);
 78:
 79:   // Calculate for pion, kaon, proton masses
 80:   const std::vector<double>       fixmass = {PDG::mpi0, PDG::mpi, PDG::mK, PDG::mp};
 81:   std::vector<std::vector<double>> dsdpt(fixmass.size(), std::vector<double>(Nbins, 0.0));
 82:   std::vector<double>              maxval(fixmass.size(), 0.0);
 83:
```



```
 84:   // ------------------------------------------------------------------------
 85:   // Very slow
 86:
 87:   // Pre-evaluate dsigma/dpt |_y=0 (mid rapidity)
 88:   for (const auto &k : indices(fixmass)) {
 89:     const double m2 = pow2(fixmass[k]);
 90:     for (std::size_t i = 0; i < Nbins; ++i) {
 91:       const double mtval = msqrt(pow2(ptval[i]) + m2);
 92:
 93:       // PDF
 94:       dsdpt[k][i] = ptval[i] * mtval * std::pow(1.0 + (q - 1.0) * mtval / T, q / (1.0 - q));
 95:       // Save maximum
 96:       maxval[k] = (dsdpt[k][i] > maxval[k]) ? dsdpt[k][i] : maxval[k];
 97:     }
 98:   }
 99:   // ------------------------------------------------------------------------
100:
101:   // For each particle, generate pt
102:   x.resize(mass.size(), 0.0);
103:   for (const auto &p : indices(mass)) {
104:     // Find the best distribution
105:     unsigned int best    = 0;
106:     double       mindist = 1e32;
107:     for (std::size_t k = 0; k < fixmass.size(); ++k) {
108:       const double dist = std::abs(fixmass[k] - mass[p]);
109:       if (dist < mindist) {
110:         mindist = dist;
111:         best    = k;
112:       }
113:     }
114:
115:     // Acceptance-Rejection, very slow
116:     unsigned int trials = 0;
117:     while (true) {
118:       const int BIN = RANDI(rng.rng);
119:       if (rng.U(0.0, 1.0) * maxval[best] < dsdpt[best][BIN]) {
120:         x[p] = ptval[BIN];
121:         break;
122:       }
123:       ++trials;
124:       if (trials > MAXTRIAL) {
125:         break;  // failed
126:       }
127:     }
128:   }
129: }
130:
131: // N-body fragmentation with tube (cylinder) phase space
132: // (approximate target distribution is flat over rapidity, pt from a given
133: // distribution)
134: //
135: // [REFERENCE: Jadach, Computer Physics Communications, 9 (1975) 297-304]
136: // [REFERENCE: UA5, http://cds.cern.ch/record/173907/files/198701320.pdf]
137: //
138: // This function contains a mixture of dynamics and kinematics, i.e.,
139: // is not a pure phase space and is suitable for soft fragmentation studies.
140: //
141: // Return: 1.0 for a valid fragmentation and -1.0 for a kinematically impossible
142:
143: double MFragment::TubeFragment(const M4Vec &mother, double M0, const std::vector<double> &m,
144:                                std::vector<M4Vec> &p, double q, double T, double lambda,
145:                                double maxpt, MRandom &rng, const std::string &pt_distribution) {
146:   // Number of re-trials in a case of failing kinematics
147:   const unsigned int MAXTRIAL = 30;
148:   unsigned int       trials   = 0;
149:
150:   const unsigned int    N = m.size();
151:   std::valarray<double> m2(N);
152:   for (const auto &i : indices(m)) { m2[i] = pow2(m[i]); }
153:   // transverse momentum px, py
154:   std::valarray<double> px(N);
155:   std::valarray<double> py(N);
156:   std::valarray<double> pt2(N);
157:
158:   // rapidity and transverse mass
159:   std::valarray<double> y(N);
160:   std::valarray<double> mt(N);
161:
162:   while (true) {
163:     // Random sample px,py and initial rapidity
164:     std::vector<double> pt(N);
165:
166:     if (pt_distribution == "powexp") {
```



```
167:       ExpPowRND(q, T, maxpt, m, pt, rng);
168:     } else if (pt_distribution == "exp") {
169:       for (const auto &i : indices(pt)) {
170:         pt2[i] = rng.ExpRandom(lambda);
171:         pt[i]  = msqrt(pt2[i]);
172:       }
173:     } else {
174:       throw std::invalid_argument("MFragment::TubeFragment: Unknown pt-distribution parameter = " +
175:                                   pt_distribution);
176:     }
177:
178:     for (const auto &i : indices(pt)) {
179:       // Sample angle, px, py
180:       const double phi = rng.U(0, 2.0 * gra::math::PI);
181:       px[i]            = pt[i] * std::cos(phi);
182:       py[i]            = pt[i] * std::sin(phi);
183:
184:       // Rapidity
185:       y[i] = rng.U(0.0, 1.0);
186:     }
187:     std::sort(begin(y), end(y));  // Rapidity ordering
188:
189:     // ------------------------------------------------------------------
190:     // Scale rapidities to span the full range [0,1] (min=0, max=N-1)
191:     y = (y - y[0]) / (y[N - 1] - y[0]);
192:
193:     // Set transverse momentum with zero sum
194:     px  = px - px.sum() / N;
195:     py  = py - py.sum() / N;
196:     pt2 = px * px + py * py;
197:
198:     // Transverse mass for all particles
199:     mt = sqrt(m2 + pt2);
200:
201:     // ------------------------------------------------------------------
202:     // Solve rapidity scale factor 'alpha'
203:     double alpha = 0.0;
204:     if (!SolveAlpha(alpha, M0, m, mt, y)) {
205:       ++trials;
206:       if (trials > MAXTRIAL) {
207:         return -1.0;
208:       } else {
209:         continue;
210:       }
211:     }
212:     // Scale all rapidities
213:     y = alpha * y;
214:
215:     // Set longitudinal momentum and energy using 'alpha'
216:     // get boost factor to +- 0
217:     const double Q                = (mt * exp(y)).sum();
218:     y                             = y + std::log(M0 / Q);
219:     const std::valarray<double> pz = mt * sinh(y);
220:     const std::valarray<double> E  = mt * cosh(y);
221:
222:     // ------------------------------------------------------------------
223:     // Set all particles 4-momenta and boost to the original frame
224:     p.resize(N);
225:     const int sign = 1;  // positive
226:     for (const auto &i : indices(p)) {
227:       p[i] = M4Vec(px[i], py[i], pz[i], E[i]);
228:       gra::kinematics::LorentzBoost(mother, M0, p[i], sign);
229:     }
230:
231:     // Check EM-conservation
232:     M4Vec p_sum(0, 0, 0, 0);
233:     for (const auto &n : p) { p_sum += n; }
234:     bool valid = gra::math::CheckEMC(p_sum - mother);
235:
236:     // Check rapidity (floating points can fail after boost in forward)
237:     for (const auto &i : indices(p)) {
238:       if (std::isnan(p[i].Rap()) || std::isinf(p[i].Rap())) {
239:         valid = false;
240:         break;
241:       }
242:     }
243:     if (!valid) {
244:       ++trials;
245:       if (trials > MAXTRIAL) {
246:         return -1.0;
247:       } else {
248:         continue;
249:       }
```

```
250:     }
251:     break;  // Fragmentation succesful!
252:   }
253:   return 1.0;
254: }
255:
256: // ------------------------------------------------------------------
257: // Solve rapidity scale factor via Newton iteration
258: // (alternative strategies are viable, too, this does not converge
259: // always with non-gaussian pt-distributions)
260: //
261: // a_1 = a_0 - f(a_0)/f'(a_1), find root a such that f(a) = 0
262: //   iteratively via
263: // a_{n+1} = a_n - f(a_n) / f'(a_n)
264: //
265: bool MFragment::SolveAlpha(double &alpha, double M0, const std::vector<double> &m,
266:                            const std::valarray<double> &mt, const std::valarray<double> &y) {
267:   const double       STOP_EPS = 1e-8;
268:   const unsigned int MAXITER  = 20;
269:
270:   // Starting value
271:   const int    N        = m.size();
272:   const double C        = std::log(M0 * M0);
273:   alpha                 = C - std::log(m[0] * m[N - 1]);
274:   std::vector<double> E = {0, 0, 0, 0};
275:
276:   unsigned int iter = 0;
277:   while (true) {
278:     const std::valarray<double> x = exp(alpha * y);
279:
280:     E[0] = (mt * x).sum();
281:     E[1] = (mt / x).sum();
282:     E[2] = (mt * y * x).sum();  // Derivative d/dy
283:     E[3] = (mt * y / x).sum();  // Derivative d/dy
284:
285:     // Iterate the solution, d/dx ln(x) = 1/x
286:     const double DY = E[0] * E[1] * (C - std::log(E[0] * E[1])) / (E[0] * E[3] - E[1] * E[2]);
287:     alpha -= DY;
288:
289:     if (std::abs(N * DY / alpha) < STOP_EPS) { return true; }
290:     if (std::isnan(alpha) || (++iter) > MAXITER) { return false; }
291:   }
292: }
293:
294: // N* decay table [set manually according to experimental data]
295: //
296: // Q is the proton / antiproton charge (1,-1)
297: // M0 is the N* mass
298: //
299: void MFragment::NstarDecayTable(int Q, double M0, std::vector<int> &pdgcodes, MRandom &rng) {
300:   int decaymode = 0;
301:
302:   // Only 2-body decay possible, mass below 3-body threshold
303:   if (M0 < (PDG::mp + 2 * PDG::mpi)) {
304:     decaymode = 0;
305:
306:     // 2- or 3-body decay possible
307:   } else {
308:     // C++11, thread_local is also static
309:     // 2->body / 3-body branching ratios from PDG
310:     thread_local std::discrete_distribution<> d({0.60, 0.40});
311:     decaymode = d(rng.rng);  // Draw random
312:   }
313:
314:   // ------------------------------------------------------------------
315:   // N*(J^P = 1/2^+) Decay Parameters from PDG
316:   // Only major decays implemented
317:
318:   std::vector<int> decayids;
319:
320:   // 2-Body channel
321:   if (decaymode == 0) {
322:     // Subchannels
323:     // C++11, thread_local is also static
324:     thread_local std::discrete_distribution<> subd(
325:         {2.0 / 3.0, 1.0 / 3.0});  // Branching ratios from Clebsch-Gordan
326:     const int channel = subd(rng.rng);
327:
328:     // PDG-ID of subchannels
329:     const std::vector<std::vector<int>> ID = {{Q * PDG_n, Q * PDG_pip},  // (anti)neutron & pi(-)+
330:                                               {Q * PDG_p, PDG_pi0}};     // (anti)proton  & pi0
331:
332:     // Choose the decay channel
```



```
333:     pdgcodes = ID[channel];
334:   }
335:
336:   // 3-body channel
337:   if (decaymode == 1) {
338:     // Subchannels
339:     // C++11, thread_local is also static
340:     thread_local std::discrete_distribution<> subd(
341:         {1.0 / 3.0, 1.0 / 3.0, 1.0 / 3.0});  // Braching ratios ansatz
342:     const int channel = subd(rng.rng);
343:
344:     // PDG-ID of subchannels
345:     const std::vector<std::vector<int>> ID = {{Q * PDG_n, Q * PDG_pip, PDG_pi0},
346:                                               {Q * PDG_p, PDG_pip, PDG_pim},
347:                                               {Q * PDG_p, PDG_pi0, PDG_pi0}};
348:
349:     // Perhaps to add
350:     //{Q*PDG_delta0, Q*PDG_pip, PDG_pi0},    // delta0 & pi+ & pi0
351:     //{Q*PDG_deltap, PDG_pip, PDG_pim},      // delta+ & pi+ & pi-
352:     //{Q*PDG_deltap, PDG_pi0, PDG_pi0}};     // delta+ & pi0 & pi0
353:
354:     // Draw the decay channel
355:     pdgcodes = ID[channel];
356:   }
357: }
358:
359: // Return excited forward proton masses
360: void MFragment::GetForwardMass(double &mass1, double &mass2, bool &excite1, bool &excite2,
361:                                unsigned int excite, MRandom &random) {
362:   if (excite == 0) {  // Fully elastic
363:     mass1   = PDG::mp;
364:     mass2   = PDG::mp;
365:     excite1 = false;
366:     excite2 = false;
367:   } else if (excite == 1) {  // Single excitation
368:     if (random.U(0, 1) < 0.5) {
369:       mass1 = PDG::mp;
370:       GetSingleForwardMass(mass2, random);
371:       excite1 = false;
372:       excite2 = true;
373:     } else {
374:       GetSingleForwardMass(mass1, random);
375:       mass2   = PDG::mp;
376:       excite1 = true;
377:       excite2 = false;
378:     }
379:   } else if (excite == 2) {  // Double excitation
380:     GetSingleForwardMass(mass1, random);
381:     GetSingleForwardMass(mass2, random);
382:     excite1 = true;
383:     excite2 = true;
384:   }
385: }
386:
387: // Return excited forward system mass
388: void MFragment::GetSingleForwardMass(double &mass, MRandom &random) {
389:   // Good-Walker resonance excitation probabilities
390:   // C++11, thread_local is also static
391:   thread_local std::discrete_distribution<> d(
392:       {PARAM_NSTAR::rrc[0], PARAM_NSTAR::rrc[1], PARAM_NSTAR::rrc[2]});
393:
394:   const int state = d(random.rng);  // Draw random
395:
396:   // Excited proton states
397:   double M0    = 0.0;
398:   double width = 0.0;
399:
400:   if (state == 0) {
401:     M0    = 1.440;  // N*
402:     width = 0.325;
403:   } else if (state == 1) {
404:     M0    = 1.680;  // N**
405:     width = 0.140;
406:   } else if (state == 2) {
407:     M0    = 2.190;  // N***
408:     width = 0.450;
409:   }
410:
411:   // Draw the excited state mass
412:   const double safe_margin = 0.01;
413:   double       mX          = 0;
414:   while (true) {
415:     mX = random.RelativisticBWRandom(M0, width, 1e6);
```



```
416:     if (mX > (PDG::mp + PDG::mpi + safe_margin)) {  // && mX < lts.sqrt_s) {
417:       mass = mX;
418:       return;
419:     }
420:   }
421: }
422:
423: // Simple statistical toy particle pick-up, nothing more
424: //
425: bool MFragment::PickParticles(double M, unsigned int N, int B, int S, int Q,
426:                               std::vector<double> &mass, std::vector<int> &pdgcode, const MPDG &PDG,
427:                               MRandom &rng) {
428:   const unsigned int MAXTRIAL = 1e4;
429:
430:   // C++11, thread_local is also static
431:   thread_local std::uniform_int_distribution<int> RAND13(0, 2);
432:
433:   // Pion, Kaon, Proton ratios [MEASURED]
434:   const double              denom3 = (1 + 0.1 + 0.06);
435:   const std::vector<double> ratio3 = {1.0 / denom3, 0.1 / denom3, 0.06 / denom3};
436:
437:   const std::vector<int> charged_pdg = {211, 321, 2212};  // pi+-, K+-, (anti)proton
438:   const std::vector<int> charged_B   = {0, 0, 1};
439:   const std::vector<int> charged_S   = {0, 1, 0};
440:
441:   const std::vector<int> neutral_pdg = {111, 311, 2112};  // pi0, K0, neutron
442:   const std::vector<int> neutral_B   = {0, 0, 1};
443:   const std::vector<int> neutral_S   = {0, 1, 0};
444:
445:   const double QProb  = 2.0 / 3.0;  // Charged probability
446:   const double DOUBLE = 0.9;        // Prob to pick particle pair
447:   // ------------------------------------------------------------------
448:
449:   // Now resize!
450:   mass.resize(N, 0);
451:   pdgcode.resize(N, 0);
452:   std::vector<int> Qcharges(N, 0);
453:   std::vector<int> Bcharges(N, 0);
454:   std::vector<int> Scharges(N, 0);
455:   unsigned int     trials = 0;
456:
457:   while (true) {
458:     unsigned int i = 0;
459:
460:     do {  // Hadron picking
461:
462:       // Charged
463:       if (rng.U(0, 1) < QProb) {
464:         // Charged pair
465:         if (rng.U(0, 1) < DOUBLE && i <= N - 2) {
466:           // Sample particle flavour
467:           while (true) {
468:             const int bin = RAND13(rng.rng);
469:
470:             if (rng.U(0, 1) < ratio3[bin]) {
471:               Qcharges[i] = 1;
472:               Bcharges[i] = charged_B[bin];
473:               Scharges[i] = charged_S[bin];
474:               pdgcode[i]  = charged_pdg[bin];
475:               ++i;  // Next slot
476:
477:               Qcharges[i] = -1;
478:               Bcharges[i] = -charged_B[bin];
479:               Scharges[i] = -charged_S[bin];
480:               pdgcode[i]  = -charged_pdg[bin];
481:               ++i;
482:               break;
483:             }
484:           }
485:
486:           // Single
487:         } else {
488:           while (true) {
489:             const int bin = RAND13(rng.rng);
490:
491:             if (rng.U(0, 1) < ratio3[bin]) {
492:               int sign = 0;
493:               sign     = (rng.U(0, 1) < 0.5) ? 1 : -1;
494:
495:               Qcharges[i] = sign;
496:               Bcharges[i] = sign * charged_B[bin];
497:               Scharges[i] = sign * charged_S[bin];
498:               pdgcode[i]  = sign * charged_pdg[bin];
```



```
499:               ++i;
500:               break;
501:             }
502:           }
503:         }
504:
505:         // Neutral
506:       } else {
507:         while (true) {
508:           // Sample particle flavour
509:           const int bin = RAND13(rng.rng);
510:           if (rng.U(0, 1) < ratio3[bin]) {
511:             Qcharges[i] = 0;
512:             Bcharges[i] = neutral_B[bin];
513:             Scharges[i] = neutral_S[bin];
514:             pdgcode[i]  = neutral_pdg[bin];
515:             ++i;
516:             break;
517:           }
518:         }
519:       }
520:     } while (i < N);  // Picking loop
521:
522:     // Sum charges
523:     const int Q_sum = std::accumulate(Qcharges.begin(), Qcharges.end(), 0);
524:     const int B_sum = std::accumulate(Bcharges.begin(), Bcharges.end(), 0);
525:     const int S_sum = std::accumulate(Scharges.begin(), Scharges.end(), 0);
526:
527:     // Get corresponding masses
528:     for (std::size_t i = 0; i < N; ++i) {
529:       // printf("PDGCODE[i=%d] = %d [N=%d] \n", i, pdgcode[i], N);
530:       mass[i] = PDG.FindByPDG(pdgcode[i]).mass;
531:     }
532:     const double M_sum = std::accumulate(mass.begin(), mass.end(), 0.0);
533:
534:     // Check charge and mass threshold
535:     bool Q_check = (Q_sum == Q) ? true : false;
536:     bool B_check = (B_sum == B) ? true : false;
537:     bool S_check = (S_sum == S) ? true : false;
538:     bool M_check = (M_sum < M) ? true : false;
539:
540:     if (Q_check && B_check && S_check && M_check) {
541:       break;  // we are ok!
542:     } else {
543:       // printf("[PICK] Q_sum = %d, B_sum = %d, M_sum = %0.1f, M = %0.1f, N = %d
544:       // \n", Q_sum, B_sum, M_sum, M, N);
545:       ++trials;
546:       if (trials > MAXTRIAL) { return false; }
547:     }
548:   }  // Out loop
549:
550:   return true;
551: }
552:
553: }  // namespace gra
554:
```

```
// Custom user cuts which cannot be implemented directly via json steering files
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

// C++
#include <complex>
#include <iostream>
#include <vector>

// Own
#include "Graniitti/MForm.h"
#include "Graniitti/MKinematics.h"
#include "Graniitti/MMath.h"
#include "Graniitti/MUserCuts.h"

namespace gra {
// USERCUTS (implement custom cuts here; cuts which cannot be implemented
// in .json steering file). Label these by unique integer.

bool UserCut(int id, const gra::LORENTZSCALAR &lts) {
  // ** NO CUTS CASE, THIS SHOULD BE FIRST **
  if (id == 0) {
    return true;
  }

  // ---------------------------------------------------------------------
  // "Spin-filter" cut ('Glueball' filter)

  // Forward proton |dpt| < 0.3 GeV
  else if (id == -3) {
    const double dpt = (lts.pfinal[1] - lts.pfinal[2]).Pt();
    if (dpt < 0.3) {
      // fine
    } else {
      return false;  // did not pass
    }
  }

  // Forward proton |dpt| > 0.3 GeV
  else if (id == 3) {
    const double dpt = (lts.pfinal[1] - lts.pfinal[2]).Pt();
    if (dpt > 0.3) {
      // fine
    } else {
      return false;  // did not pass
    }
  }

  // ---------------------------------------------------------------------
  // "Spin-filter" cut

  // Forward proton pt1 dot pt2 < 0
  else if (id == -111) {
    if (lts.pfinal[1].DotPt(lts.pfinal[2]) < 0) {
      // fine
    } else {
      return false;  // did not pass
    }
  }

  // Forward proton pt1 dot pt2 > 0
  else if (id == 111) {
    if (lts.pfinal[1].DotPt(lts.pfinal[2]) > 0) {
      // fine
    } else {
      return false;  // did not pass
    }
  }


  // ---------------------------------------------------------------------
  // "Spin-filter" cut

  // Forward proton |deltaphi| in (90, 180]
  else if (id == 90180) {
    const double deltaphiabs = lts.pfinal[1].DeltaPhiAbs(lts.pfinal[2]);
    if (gra::math::Deg2Rad(90) < deltaphiabs && deltaphiabs <= gra::math::Deg2Rad(180)) {
      // fine
    } else {
      return false;  // did not pass
    }
  }
```



```

  // Forward proton |deltaphi| in (0, 90]
  else if (id == 90) {
    const double deltaphiabs = lts.pfinal[1].DeltaPhiAbs(lts.pfinal[2]);
    if (0 < deltaphiabs && deltaphiabs <= gra::math::Deg2Rad(90)) {
      // fine
    } else {
      return false;  // did not pass
    }
  }

  // ---------------------------------------------------------------------
  // STAR/RHIC \sqrt(s) = 200 GeV pi+pi- (+ other cuts needed in .json file)
  // indico.cern.ch/event/713101/contributions/3102315/
  // attachments/1705771/2748440/Diffraction2018_RafalSikora.pdf
  //
  else if (id == 280818) {
    // Loop over forward protons
    std::vector<int> indices = {1, 2};

    for (const auto &i : indices) {
      if (gra::math::pow2(lts.pfinal[i].Px() + 0.3) + gra::math::pow2(lts.pfinal[i].Py()) <
          0.25) {  // GeV^2
                   // fine
      } else {
        return false;  // not passed
      }

      if (0.2 < std::abs(lts.pfinal[i].Py()) && std::abs(lts.pfinal[i].Py()) < 0.4) {  // GeV
                                                                                       // fine
      } else {
        return false;  // not passed
      }

      if (lts.pfinal[i].Px() > -0.2) {  // GeV
                                        // fine
      } else {
        return false;  // not passed
      }
    }

    // [arxiv.org/abs/1608.03765]

  } else if (id == 160803765) {
    const double xi1 = (lts.pbeam1.Pz() - lts.pfinal[1].Pz()) / lts.pbeam1.Pz();
    const double xi2 = (lts.pbeam2.Pz() - lts.pfinal[2].Pz()) / lts.pbeam2.Pz();

    const double XI_MAX = 0.03;

    if (xi1 < XI_MAX && xi2 < XI_MAX) {
      // fine
    } else {
      return false;  // not passed
    }
  }

  // ---------------------------------------------------------------------
  // CDF exclusive dijets
  // [arxiv.org/abs/0712.0604]

  else if (id == 7120604) {
    // antiproton longitudinal momentum loss fraction
    const double xi_pbar = (lts.pbeam2.Pz() - lts.pfinal[2].Pz()) / lts.pbeam2.Pz();

    if (0.03 < xi_pbar && xi_pbar < 0.08) {
      // fine
    } else {
      return false;  // not passed
    }
  }

  // ---------------------------------------------------------------------
  // ATLAS yy->mu+mu- 13 TeV fiducial cuts (+other cuts needed in .json file)
  // [arxiv.org/abs/hep-ex/1708.04053]

  else if (id == 170804053) {
    const double M = gra::math::msqrt(lts.m2);

    if (12 <= M && M < 30) {  // GeV
      if (lts.decaytree[0].p4.Pt() > 6 && lts.decaytree[1].p4.Pt() > 6) {
        // fine
      } else {
        return false;  // not passed
```



```
      }
    } else if (30 <= M && M <= 70) {  // GeV
      if (lts.decaytree[0].p4.Pt() > 10 && lts.decaytree[1].p4.Pt() > 10) {
        // fine
      } else {
        return false;  // not passed
      }
    }
  }

  // ---------------------------------------------------------------------
  // ATLAS pi+pi- 13 TeV roman pot fiducial cuts
  // (+other cuts needed in .json file)
  // from R. Sikora, ATLAS poster, Bad Honnef QCD School 2017
  //
  // (N.B. check the implementation
  // c.f. cut |t| > 0.03 GeV^2 seems to give more physical results)

  else if (id == 1230123) {
    // Forward protons |py| and |phi|
    std::vector<double> pyabs  = {std::abs(lts.pfinal[1].Py()), std::abs(lts.pfinal[2].Py())};
    std::vector<double> phiabs = {std::abs(lts.pfinal[1].Phi()), std::abs(lts.pfinal[2].Phi())};
    for (std::size_t i = 0; i < 2; ++i) {
      if ((0.17 < pyabs[i]) && (pyabs[i] < 0.5)) {  // GeV
                                                    // fine
      } else {
        return false;  // not passed
      }
      if ((gra::math::PI / 4 < phiabs[i]) && (phiabs[i] < 3.0 * gra::math::PI / 4)) {
        // fine
      } else {
        return false;  // not passed
      }
    }

    // Roman pot geometry
    const double deltaphiabs = lts.pfinal[1].DeltaPhiAbs(lts.pfinal[2]);
    if ((deltaphiabs < gra::math::Deg2Rad(40.0)) || (deltaphiabs > gra::math::Deg2Rad(140.0))) {
      // fine
    } else {
      return false;  // not passed
    }
  } else {  // Poor input
    std::string str = "MUserCuts:: Unknown cut ID: " + std::to_string(id);
    throw std::invalid_argument(str);
  }

  return true;
}
}  // namespace gra
```



```
// Functional methods for Spherical Harmonic Expansions
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

// C++
#include <complex>
#include <iostream>
#include <vector>

// Own
#include "Graniitti/MAux.h"
#include "Graniitti/MMath.h"
#include "Graniitti/MMatrix.h"
#include "Graniitti/MRandom.h"
#include "Graniitti/MSpherical.h"

// Libraries
#include "rang.hpp"

using gra::aux::indices;
using gra::math::PI;
using gra::math::msqrt;
using gra::math::pow2;
using gra::math::zi;

namespace gra {
namespace spherical {
// Monte Carlo integral I ~= V 1/N \sum_{i=1}^N f(\vec{x}_i) = V <f(x)>
// True integral being  I = \int_\Omega f(\vec{x}) d\vec{x}
//
// This inner-product (overlap integral) matrix is identity if the phase space
// is flat (uniform). In a geometrically restricted phase space, these basis
// functions get mixed => lost orthogonality.

MMatrix<double> GetGMixing(const std::vector<Omega> &events, const std::vector<std::size_t> &ind,
                           int LMAX, const std::string &mode) {
  const int NCOEF = (LMAX + 1) * (LMAX + 1);

  std::cout << "GetGMixing: mode = " << mode << std::endl;
  std::cout << "Generated flat MC phase space events = " << ind.size() << std::endl;

  // Construct the efficiency coefficients EPSILON_LM with linear
  // indexing
  MMatrix<double> E(NCOEF, NCOEF, 0.0);
  MMatrix<double> E2(NCOEF, NCOEF, 0.0);  // for the uncertainty

  // Loop over GENERATED MC events and do the integral in effect via
  // uniform MC sampling
  // We evaluate the integral:
  //
  // eta_LM = \int eta(Omega) Re Y_LM(Omega) dOmega, Omega = (costheta,phi)
  int fiducial = 0;
  int selected = 0;

  const double VOL = 4.0 * PI;  // [costheta] x [phi] plane area

  for (const auto &k : ind) {
    const bool fid = events[k].fiducial;
    const bool sel = events[k].selected;

    // Flat phase space
    if (mode == "fla") {
      // all fine
    }
    // Geometric acceptance ("Fiducial phase space")
    else if (mode == "fid") {
      if (!fid) {
        continue;
      } else {
        ++fiducial;
      }
    }
    // Geometric x Efficiency ("Detector level")
    else if (mode == "det") {
      if (!fid) {
        continue;
      } else {
        ++fiducial;
      }
      if (!sel) {
        continue;
      } else {
```

```
84:         ++selected;
85:       }
86:     } else {
87:       throw std::invalid_argument("spherical::GetELM: Unknown mode " + mode);
88:     }
89:
90:     const double costheta = events[k].costheta;
91:     const double phi      = events[k].phi;
92:
93:     for (int l = 0; l <= LMAX; ++l) {
94:       for (int m = -l; m <= l; ++m) {
95:         const int index = LinearInd(l, m);
96:
97:         const std::complex<double> Y   = gra::math::Y_complex_basis(costheta, phi, l, m);
98:         const double               ReY = gra::math::NReY(Y, l, m);
99:
100:        for (int lprime = 0; lprime <= LMAX; ++lprime) {
101:          for (int mprime = -lprime; mprime <= lprime; ++mprime) {
102:            const int                  indexprime = LinearInd(lprime, mprime);
103:            const std::complex<double> Yprime =
104:                gra::math::Y_complex_basis(costheta, phi, lprime, mprime);
105:            const double ReYprime = gra::math::NReY(Yprime, lprime, mprime);
106:
107:            // Sum to the MC integral and its squared version
108:            // (for uncertainty)
109:            const double f = VOL * ReY * ReYprime;  // Note volume term
110:            E[index][indexprime] += f;
111:            E2[index][indexprime] += f * f;
112:          }
113:        }
114:      }
115:    }
116:  }
117:
118:  if (mode == "fid" || mode == "det") {
119:    printf(
120:        "Fiducial flat MC phase space events = %d (acceptance %0.3f "
121:        "percent) \n",
122:        fiducial, fiducial / static_cast<double>(ind.size()) * 100);
123:  }
124:  if (mode == "det") {
125:    printf(
126:        "Selected flat MC phase space events = %d (efficiency %0.3f "
127:        "percent) \n",
128:        selected, selected / static_cast<double>(fiducial) * 100);
129:  }
130:   if (mode == "fid" || mode == "det") {
131:     if (fiducial < 50 || selected < 50) {
132:       std::cout << rang::fg::red << "GetGMixing:: Too low MC event count in the mass bin!"
133:                 << rang::fg::reset << std::endl;
134:     }
135:   }
136:   std::cout << std::endl;
137:
138:   // Do the normalization
139:   const double N_generated =
140:       static_cast<double>(ind.size());  // Generated events within this mass interval
141:   std::vector<double> E_error(E.size_row(), 0.0);
142:   std::cout << "Symmetric mixing matrix integral coefficients G_{ll'}^{mm'}:" << std::endl;
143:   std::vector<double> rowsum(NCOEF, 0.0);
144:   std::cout << "          " << std::endl;
145:
146:   for (int l = 0; l <= LMAX; ++l) {
147:     for (int m = -l; m <= l; ++m) { printf("G_%d%d  \t", l, m); }
148:   }
149:   std::cout << std::endl;
150:
151:   for (int l = 0; l <= LMAX; ++l) {
152:     for (int m = -l; m <= l; ++m) {
153:       const int index = LinearInd(l, m);
154:
155:       printf("G_%d%d : ", l, m);
156:       double colsum = 0;
157:
158:       for (int lprime = 0; lprime <= LMAX; ++lprime) {
159:         for (int mprime = -lprime; mprime <= lprime; ++mprime) {
160:           const int indexprime = LinearInd(lprime, mprime);
161:
162:           E[index][indexprime] /= N_generated;
163:           E2[index][indexprime] /= N_generated;
164:           colsum += E[index][indexprime];
165:
166:           double error = CalcError(E2[index][indexprime], E[index][indexprime], N_generated);
```



```
167:           printf("%6.3f +- %6.3f \t", E[index][indexprime], error);
168:         }
169:       }
170:       printf(" | %6.3f \n", colsum);
171:     }
172:   }
173:
174:   gra::aux::PrintBar("-");
175:   std::cout << "     " << std::endl;
176:   for (std::size_t j = 0; j < E.size_row(); ++j) {
177:     double rowsum = 0;
178:     for (std::size_t i = 0; i < E.size_col(); ++i) { rowsum += E[i][j]; }
179:     printf("%6.3f\t", rowsum);
180:   }
181:   printf("\n\n\n\n");
182:
183:   // Calculate matrix condition number via SVD
184:   Eigen::MatrixXd                A = gra::aux::MatrixZEigen(E);
185:   Eigen::JacobiSVD<Eigen::MatrixXd> svd(A);
186:   Eigen::MatrixXd                sval = svd.singularValues();
187:
188:   std::cout << "SVD singular values of the matrix:" << std::endl;
189:   for (std::size_t i = 0; i < E.size_row(); ++i) { printf("%0.4f ", sval(i)); }
190:   std::cout << std::endl;
191:   const double conditionnumber = sval(0) / sval(E.size_row() - 1);
192:   if (conditionnumber < 10) {
193:     std::cout << rang::fg::green;
194:   } else {
195:     std::cout << rang::fg::red;
196:   }
197:
198:   printf("Condition number: sMax/sMin = %0.4f \n", conditionnumber);
199:   std::cout << std::endl << std::endl << std::endl;
200:   std::cout << rang::fg::reset;
201:   return E;
202: }
203:
204: // Acceptance expansion coefficients using MC events
205: std::pair<std::vector<double>, std::vector<double>> GetELM(const std::vector<Omega> &      MC,
206:                                                            const std::vector<std::size_t> &ind,
207:                                                            int LMAX, const std::string &mode) {
208:   const int NCOEF = (LMAX + 1) * (LMAX + 1);
209:
210:   std::cout << "GetELM: mode = " << mode << std::endl;
211:   std::cout << "Generated flat MC phase space events = " << ind.size() << std::endl;
212:
213:   // Construct the efficiency coefficients E_LM with linear indexing
214:   std::vector<double> E(NCOEF, 0.0);
215:   std::vector<double> E2(NCOEF, 0.0);  // for the uncertainty
216:
217:   // Loop over GENERATED MC events and do the integral in effect via
218:   // uniform MC sampling
219:   // We evaluate the integral:
220:   //
221:   // E_LM = \int E(Omega) Re Y_LM(Omega) dOmega, Omega = (costheta,phi)
222:
223:   int          fiducial = 0;
224:   int          selected = 0;
225:   const double V        = sqrt(4.0 * PI);  // Normalization volume
226:
227:   for (const auto &k : ind) {
228:     bool fid = MC[k].fiducial;
229:     bool sel = MC[k].selected;
230:
231:     // Flat flat phase space
232:     if (mode == "fla") {
233:       // all fine
234:     }
235:     // Geometric acceptance ("Fiducial phase space")
236:     else if (mode == "fid") {
237:       if (!fid) {
238:         continue;
239:       } else {
240:         ++fiducial;
241:       }
242:     }
243:     // Geometric x Efficiency ("Detector level")
244:     else if (mode == "det") {
245:       if (!fid) {
246:         continue;
247:       } else {
248:         ++fiducial;
249:       }
```



```
250:       if (!sel) {
251:         continue;
252:       } else {
253:         ++selected;
254:       }
255:     } else {
256:       throw std::invalid_argument("spherical::GetELM: Unknown mode " + mode);
257:     }
258:
259:     const double costheta = MC[k].costheta;
260:     const double phi      = MC[k].phi;
261:
262:     for (int l = 0; l <= LMAX; ++l) {
263:       for (int m = -l; m <= l; ++m) {
264:         const std::complex<double> Y = gra::math::Y_complex_basis(costheta, phi, l, m);
265:
266:         // Sum to the MC integral and its square (for uncertainty)
267:         double f = V * gra::math::NReY(Y, l, m);  // Note volume V term
268:
269:         const int index = LinearInd(l, m);
270:         E[index] += f;
271:         E2[index] += pow2(f);
272:       }
273:     }
274:   }
275:   if (mode == "fid" || mode == "det") {
276:     printf(
277:         "Fiducial flat MC phase space events = %d (geometric-kinematic "
278:         "acceptance "
279:         "%0.3f percent) \n\n",
280:         fiducial, fiducial / static_cast<double>(ind.size()) * 100);
281:   }
282:   if (mode == "det") {
283:     printf(
284:         "Selected flat MC phase space events = %d (fiducial efficiency "
285:         "%0.3f percent) "
286:         "\n\n",
287:         selected, selected / static_cast<double>(fiducial) * 100);
288:   }
289:   if (mode == "fid") {
290:     if (fiducial < 1000) {
291:       std::cout << "GetELM:: Very low [fiducial] MC event count = " << fiducial
292:                 << " in the mass bin!" << std::endl;
293:     }
294:   }
295:   if (mode == "det") {
296:     if (selected < 1000) {
297:       std::cout << "GetELM:: Very low [selected] MC event count = " << selected
298:                 << " in the mass bin!" << std::endl;
299:     }
300:   }
301:
302:   // Do the normalization
303:   const double        N_generated = (double)ind.size();  // Generated events within this hypercell
304:   std::vector<double> E_error(E.size(), 0.0);
305:
306:   std::cout << "Acceptance decomposition coefficients:" << std::endl;
307:   double sum = 0;
308:   for (int l = 0; l <= LMAX; ++l) {
309:     for (int m = -l; m <= l; ++m) {
310:       const int index = LinearInd(l, m);
311:
312:       E[index] /= N_generated;
313:       E2[index] /= N_generated;
314:
315:       // 1 sigma MC integration uncertainty
316:       const double error = msqrt((E2[index] - std::pow(E[index], 2)) / N_generated);
317:       E_error[index]     = error;
318:       sum += E[index];
319:
320:       printf("E_%d%d \t= %12.8f +- %12.8f   (rel.error %9.3f percent) \n", l, m, E[index], error,
321:              std::abs(error / E[index]) * 100);
322:     }
323:   }
324:   printf("SUM_lm =   %0.8f \n", sum);
325:   std::cout << "Uncertainties by Monte Carlo errors = sqrt(<f^2> - <f>^2) / n)" << std::endl;
326:   std::cout << std::endl;
327:
328:   return {E, E_error};
329: }
330:
331: // Calculate expansion coefficients directly via algebraic expansion
332: //
```



```
333: std::vector<double> SphericalMoments(const std::vector<Omega> &      input,
334:                                       const std::vector<std::size_t> &ind, int LMAX,
335:                                       const std::string &mode) {
336:   const double        V     = sqrt(4.0 * PI);  // Normalization volume
337:   const unsigned int  NCOEF = (LMAX + 1) * (LMAX + 1);
338:   std::vector<double> t(NCOEF, 0.0);
339:
340:   for (int l = 0; l <= LMAX; ++l) {
341:     for (int m = -l; m <= l; ++m) {
342:       double       sum   = 0;
343:       const unsigned int index = LinearInd(l, m);
344:       for (const auto &k : ind) {
345:         // Check is this event selected by cuts
346:         const bool fidtrue = input[k].fiducial;
347:         const bool seltrue = input[k].selected;
348:
349:         // Flat flat phase space
350:         if (mode == "fla") {
351:           // all fine
352:         }
353:         // Geometric acceptance ("Fiducial level")
354:         else if (mode == "fid") {
355:           if (!fidtrue) { continue; }
356:         }
357:         // Geometric x Efficiency ("Detector level")
358:         else if (mode == "det") {
359:           if (!fidtrue) { continue; }
360:           if (!seltrue) { continue; }
361:         } else {
362:           throw std::invalid_argument("spherical::SphericalMoments: Unknown mode " + mode);
363:         }
364:
365:         const double               costheta = input[k].costheta;
366:         const double               phi      = input[k].phi;
367:         const std::complex<double> Y        = gra::math::Y_complex_basis(costheta, phi, l, m);
368:
369:         sum += gra::math::NReY(Y, l, m);
370:       }
371:       t[index] = V * sum;  // Note volume V term
372:     }
373:   }
374:   return t;
375: }
376:
377: // Calculate Y_lm values for each event
378: MMatrix<double> YLM(const std::vector<Omega> &events, int LMAX) {
379:   std::cout << "YLM:" << std::endl;
380:
381:   const unsigned int NCOEFF = (LMAX + 1) * (LMAX + 1);
382:   MMatrix<double>    Y_lm(events.size(), NCOEFF, 0.0);
383:
384:   // Loop over events
385:   for (const auto &k : indices(events)) {
386:     // Get event (costheta, phi)
387:     const double costheta = events[k].costheta;
388:     const double phi      = events[k].phi;
389:
390:     // Loop over (l,m) coefficients
391:     for (int l = 0; l <= LMAX; ++l) {
392:       for (int m = -l; m <= l; ++m) {
393:         const std::complex<double> Y = gra::math::Y_complex_basis(costheta, phi, l, m);
394:
395:         // Save its value
396:         const int index = LinearInd(l, m);
397:         Y_lm[k][index]  = gra::math::NReY(Y, l, m);
398:       }
399:     }
400:   }
401:   return Y_lm;
402: }
403:
404: // Spherical harmonic dot product between coefficients gives the integral.
405: // This is the property of orthonormality.
406: // \int G(Omega) x(omega) dOmega <=> \sum_i G_i x_i
407: double HarmDotProd(const std::vector<double> &G, const std::vector<double> &x,
408:                    const std::vector<bool> &ACTIVE, int LMAX) {
409:   double sum = 0.0;
410:   for (int l = 0; l <= LMAX; ++l) {
411:     for (int m = -l; m <= l; ++m) {
412:       const int index = LinearInd(l, m);
413:       if (ACTIVE[index]) { sum += G[index] * x[index]; }
414:     }
415:   }
```

```
416:   return sum;
417: }
418:
419: // TODO: add correlations in to the sum
420: double HarmDotProdError() { return 0.0; }
421:
422: void PrintOutMoments(const std::vector<double> &x, const std::vector<double> &x_error,
423:                      const std::vector<bool> &ACTIVE, int LMAX) {
424:   for (int l = 0; l <= LMAX; ++l) {
425:     for (int m = -l; m <= l; ++m) {
426:       const unsigned int index = LinearInd(l, m);
427:       printf("t_%d%d \t= %8.1f +- %5.1f   ", l, m, x[index], x_error[index]);
428:
429:       if (ACTIVE[index]) {
430:         std::cout << "[" << rang::fg::green << "active" << rang::fg::reset << "]" << std::endl;
431:       } else {
432:         std::cout << "[" << rang::fg::red << "inactive" << rang::fg::reset << "]" << std::endl;
433:       }
434:     }
435:   }
436:   std::cout << std::endl;
437: }
438:
439: // Calculate indices for this interval
440: //
441: std::vector<std::size_t> GetIndices(const std::vector<Omega> &events, const std::vector<double> &M,
442:                                     const std::vector<double> &Pt, const std::vector<double> &Y) {
443:   std::vector<std::size_t> ind;
444:
445:   for (const auto &i : indices(events)) {
446:     if (events[i].M > M[0] && events[i].M < M[1] && events[i].Pt > Pt[0] && events[i].Pt < Pt[1] &&
447:         events[i].Y > Y[0] && events[i].Y < Y[1]) {
448:       ind.push_back(i);
449:     }
450:   }
451:
452:   gra::aux::PrintBar("-");
453:   std::cout << rang::fg::green;
454:   printf(
455:       "MASS RANGE: [%0.3f, %0.3f] GeV, PT RANGE: [%0.3f, %0.3f] GeV, Y "
456:       "RANGE: [%0.3f, %0.3f] "
457:       ": Events in this hyperbin %lu/%lu \n\n",
458:       M[0], M[1], Pt[0], Pt[1], Y[0], Y[1], ind.size(), events.size());
459:   std::cout << rang::fg::reset;
460:
461:   return ind;
462: }
463:
464: // TestIntegrals spherical harmonics
465: // This is a test function, all integrals should give 1 if the normalization is
466: // correct
467: void TestSphericalIntegrals(int LMAX) {
468:   MRandom                 random;
469:   const unsigned int NCOEF = (LMAX + 1) * (LMAX + 1);
470:   std::cout << "TestSphericalIntegrals: " << std::endl << std::endl;
471:
472:   // Construct the efficiency coefficients EPSILON_LM with linear indexing
473:   std::vector<double> E(NCOEF, 0.0);
474:   std::vector<double> E2(NCOEF, 0.0);  // for the uncertainty
475:
476:   // We MC evaluate the integral:
477:   // \int |Re Y_LM(Omega)|^2 dOmega, Omega = (costheta,phi)
478:   const double N = 100000;  // Number of samples
479:
480:   // Integral domain volume: |costheta| x |phi| = [-1,1] x [0,2pi)
481:   const double V = 4.0 * gra::math::PI;
482:
483:   for (std::size_t k = 0; k < N; ++k) {
484:     const double costheta = random.U(-1.0, 1.0);
485:     const double phi      = random.U(0.0, 2.0 * PI);
486:
487:     for (int l = 0; l <= LMAX; ++l) {
488:       for (int m = -l; m <= l; ++m) {
489:         const std::complex<double> Y = gra::math::Y_complex_basis(costheta, phi, l, m);
490:
491:         // Function value
492:         double f = std::pow(std::abs(gra::math::NReY(Y, l, m)), 2) * V;
493:
494:         // Sum to the MC integral and its squared version (for uncertainty)
495:         const int ind = LinearInd(l, m);
496:         E[ind] += f;
497:         E2[ind] += pow2(f);
498:       }
```



```
499:     }
500:   }
501:
502:   for (int l = 0; l <= LMAX; ++l) {
503:     for (int m = -l; m <= l; ++m) {
504:       const int ind = LinearInd(l, m);
505:       E[ind] /= N;
506:       E2[ind] /= N;
507:
508:       // 1 sigma MC integration uncertainty
509:       double error = CalcError(E2[ind], E[ind], N);
510:       printf("I_%d%d \t= %12.8f +- %12.8f   (rel.error %9.3f percent) \n", l, m, E[ind], error,
511:              std::abs(error / E[ind]) * 100);
512:     }
513:   }
514:   std::cout << std::endl;
515: }
516:
517: // Find out linear index
518: int LinearInd(int l, int m) { return l * (l + 1) + m; }
519:
520: // Standard error
521: double CalcError(double f2, double f, double N) { return sqrt((f2 - std::pow(f, 2)) / N); }
522:
523: // Print matrix to a file
524: void PrintMatrix(FILE *fp, const std::vector<std::vector<double>> &A) {
525:   // Print out coefficients
526:   for (std::size_t i = 0; i < A.size(); ++i) {
527:     for (std::size_t j = 0; j < A[i].size(); ++j) { fprintf(fp, "%0.1f ", A[i][j]); }
528:     fprintf(fp, "\n");
529:   }
530:   fprintf(fp, "\n");
531: }
532:
533: // Synthesize distributions with real SH-basis
534: //
535: // f(theta,phi) = \sum_{l=0}\sum_{m=-l}^l c_lm x Y_R^{lm}(theta,phi)
536: //
537: MMatrix<double> Y_real_synthesize(const std::vector<double> &c_lm, const std::vector<bool> &ACTIVE,
538:                                   std::size_t N, std::vector<double> &costheta,
539:                                   std::vector<double> &phi, bool normalized) {
540:   const int LMAX = msqrt(ACTIVE.size()) - 1;
541:
542:   // Cos(theta) and phi
543:   costheta = math::linspace(-1.0, 1.0, N);
544:   phi      = math::linspace(-math::PI, math::PI, N);
545:
546:   // Do the expansion
547:   MMatrix<double> Z(N, N, 0.0);
548:
549:   for (int l = 0; l <= LMAX; ++l) {
550:     for (int m = -l; m <= l; ++m) {
551:       const int index = gra::spherical::LinearInd(l, m);
552:       if (!ACTIVE[index]) { continue; }  // Not active
553:
554:       for (const std::size_t &i : indices(costheta)) {
555:         for (const std::size_t &j : indices(phi)) {
556:           // Add value
557:           Z(i)[j] += c_lm[index] * math::Y_real_basis(costheta[i], phi[j], l, m);
558:         }
559:       }
560:     }
561:   }
562:
563:   if (normalized) {
564:     double max = 1e-32;
565:     for (std::size_t i = 0; i < Z.size_row(); ++i) {
566:       for (std::size_t j = 0; j < Z.size_col(); ++j) {
567:         if (Z(i)[j] > max) { max = Z(i)[j]; }
568:       }
569:     }
570:     Z = Z * (1.0 / max);  // Normalize elements
571:   }
572:
573:   return Z;
574: }
575:
576: // Vector taylor expanded error propagation from x via A to y
577: // x contains standard deviations
578: // is the Jacobi matrix
579: std::vector<double> ErrorProp(const MMatrix<double> &A, const std::vector<double> &x) {
580:   std::vector<double> y(A.size_row(), 0.0);
581:
```



```
582:   for (std::size_t m = 0; m < A.size_row(); ++m) {
583:     for (std::size_t n = 0; n < A.size_col(); ++n) { y[m] += pow2(A[m][n] * x[n]); }
584:     y[m] = msqrt(y[m]);
585:   }
586:   return y;
587: }
588:
589: }  // namespace spherical
590: }  // namespace gra
591:
```



```
  1: // Spin polarization functions
  2: //
  3: // (Wigner-D functions, Clebsch-Gordans, Arbitrary Spin-Density matrix)
  4: //
  5: //
  6: // In general, functions here use helicities for fermions which are
  7: // not normalized by the spin vector norm.
  8: // That is, functions use [-1/2, 1/2] not for example [-1, 1].
  9: //
 10: //
 11: // TBD: Add permutation coherent spin chains with Bose-Einstein / Fermi-Dirac sign
 12: // combinations, together with Breit-Wigner weights at amplitude level.
 13: //
 14: //
 15: // (c) 2017-2020 Mikael Mieskolainen
 16: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
 17:
 18: // C++ standard
 19: #include <algorithm>
 20: #include <complex>
 21: #include <iostream>
 22: #include <random>
 23: #include <vector>
 24:
 25: // Eigen
 26: #include <Eigen/Dense>
 27:
 28: // Own
 29: #include "Granitti/MAux.h"
 30: #include "Granitti/MForm.h"
 31: #include "Granitti/MMath.h"
 32: #include "Granitti/MMatrix.h"
 33: #include "Granitti/MSpin.h"
 34:
 35: using gra::math::msqrt;
 36: using gra::math::pow2;
 37: using gra::math::zi;
 38:
 39: namespace gra {
 40: namespace spin {
 41:
 42:
 43: // Used for checking eigenvalues
 44: const double epsilon = 1e-6;
 45:
 46: // Initialize helicity decay amplitude matrix T (2s1 + 1) x (2s2 + 1),
 47: //
 48: // where s1, s2 are daughter spins (0, 1/2, 1, ...)
 49: //
 50: // T_{\lambda_1,\lambda_2}
 51: //   = \sum_{ls} \alpha_{ls} x
 52: //           < J\lambda|ls0\lambda> <s\lambda|s1s2\lambda1,-\lambda2>
 53: //
 54: void InitTMatrix(gra::HELMatrix &hc, const gra::MParticle &p, const gra::MParticle &p1,
 55:                  const gra::MParticle &p2) {
 56:   const double J = p.spinX2 / 2.0;
 57:   const int    P = p.P;
 58:
 59:   const double s1 = p1.spinX2 / 2.0;
 60:   const int    P1 = p1.P;
 61:
 62:   const double s2 = p2.spinX2 / 2.0;
 63:   const int    P2 = p2.P;
 64:
 65:   // Boson or Fermion
 66:   const bool is_boson1 = (cint(s1)) ? true : false;
 67:   const bool is_boson2 = (cint(s2)) ? true : false;
 68:   const bool identical = (p1.pdg == p2.pdg) ? true : false;  // e.g. pi0 pi0 vs pi+pi-
 69:
 70:   // ------------------------------------------------------------------------
 71:   // Helicity decay amplitude matrix
 72:
 73:   hc.T = MMatrix<std::complex<double>>(static_cast<unsigned int>(2 * s1 + 1),
 74:                                        static_cast<unsigned int>(2 * s2 + 1), 0.0);
 75:
 76:   MMatrix<bool> need_to_set(20, 20, false);
 77:   MMatrix<bool> active(20, 20, false);
 78:
 79:   bool         tag_set = false;
 80:   unsigned int nonzero = 0;
 81:
 82:   // Construct T-matrix (2s1 + 1) x (2s2 + 1) elements
 83:   std::cout << std::endl;
```

```
  std::cout << "gra::spin:: "
            << " Particle 1: " << (is_boson1 ? "boson" : "fermion")
            << " , Particle 2: " << (is_boson2 ? "boson" : "fermion") << std::endl;

  if (hc.P_conservation) {
    std::cout << "Parity conservation: P = P1 x P2 x (-1)^l  [P = " << P << ", P1 = " << P1
              << ", P2 = " << P2 << "]" << std::endl;
  }
  std::cout << std::endl;
  std::cout
      << "gra::spin::InitTMatrix: Calculating SU(2) decomposition [lambda = lambda1 - lambda2]: "
      << std::endl;

  // Two loops, first check that sum |alpha_ls|^2 == 1 for active ls values

  for (int MODE = 0; MODE < 2; ++MODE) {
    for (int s = 0; s <= static_cast<int>(s1 + s2); ++s) {
      for (int l = 0; l <= static_cast<int>(J + s); ++l) {
        for (int i = 0; i < static_cast<int>(2 * s1 + 1); ++i) {
          for (int j = 0; j < static_cast<int>(2 * s2 + 1); ++j) {
            const double lambda1 = -s1 + static_cast<double>(i);  // From negative to positive
            const double lambda2 = -s2 + static_cast<double>(j);  // From negative to positive

            // Jacob-Wick
            const double lambda = lambda1 - lambda2;

            // Check angular momentum conservation for z-axis (
            if (!(abs(lambda1 - lambda2) <= J)) { continue; }  // Not conserved

            // ------------------------------------------------------------
            // Re-coupling coefficients << ... >>

            // Normalization
            const double NORM = msqrt((2.0 * l + 1.0) / (2.0 * J + 1.0));

            // <J \lambda | ls0\lambda>
            const double cg1 = gra::spin::ClebschGordan(
                static_cast<double>(l), static_cast<double>(s), 0.0, lambda, J, lambda);

            // <s \lambda | s1s2\lambda1,-\lambda2>
            const double cg2 =
                gra::spin::ClebschGordan(s1, s2, lambda1, -lambda2, static_cast<double>(s), lambda);

            // ------------------------------------------------------------

            // Angular momentum conserved (should be, by construction here)
            if (!(std::abs(lambda) <= J)) { continue; }  // not conserved

            // If parity conserving decay
            if (hc.P_conservation) {
              // Parity of the final state (hold for both
              // bosons/fermions)
              const int P_tot = P1 * P2 * std::pow(-1, l);
              if (P_tot != P) { continue; }  // not conserved
            }

            // Bose-Symmetry for identical Boson-Pairs (symmetric
            // wavefunction)
            if (is_boson1 && is_boson2 && identical) {
              if (!BoseSymmetry(l, s)) { continue; }  // not right
            }

            // Fermi-Symmetry for identical Fermion-Pairs (antisymmetric
            // wavefunction)
            if (!is_boson1 && !is_boson2 && identical) {
              if (!FermiSymmetry(l, s)) { continue; }  // not right
            }

            // CG-coupling product is non-zero
            if (std::abs(cg1 * cg2) < 1e-6) { continue; }

            // ** This coefficient is active **
            active[l][s] = true;

            // INITIALIZATION MODE
            if (MODE == 0) {
              // Has not been set, throw error next
              if (hc.alpha_set[l][s] == false) {
                need_to_set[l][s] = true;
                tag_set           = true;
              }
              ++nonzero;
            }
```



```

            // FINAL MODE
            else if (MODE == 1) {
              // T matrix element
              hc.T[i][j] += hc.alpha[l][s] * NORM * cg1 * cg2;
              printf(
                  "[ls=%d,s=%d] lambda1 = %4.1f, lambda2 = %4.1f : cg1*cg2 = %6.3f  <alpha_ls = "
                  "(%0.3f, %0.3f)>\n",
                  l, s, lambda1, lambda2, cg1 * cg2, std::real(hc.alpha[l][s]),
                  std::imag(hc.alpha[l][s]));
            }

          }  // lambda2
        }    // lambda1
      }      // l
    }        // s

    // Check normalization
    double sum_alphasq = 0;
    if (MODE == 0) {
      for (std::size_t l = 0; l < hc.alpha.size_row(); ++l) {
        for (std::size_t s = 0; s < hc.alpha.size_col(); ++s) {
          if (active[l][s] == true) { sum_alphasq += math::abs2(hc.alpha[l][s]); }
        }
      }
    }

    // Now normalize the user alpha_ls input if needed such that: sum_(ls) |alpha_ls|^2 = 1
    if (MODE == 0 && std::abs(sum_alphasq - 1.0) > 1e-6) {
      std::cout << "Renormalizing input: sum_{ls} |alpha_{ls}|^2 = " << sum_alphasq << " => 1"
                << std::endl
                << std::endl;

      for (std::size_t l = 0; l < hc.alpha.size_row(); ++l) {
        for (std::size_t s = 0; s < hc.alpha.size_col(); ++s) {
          if (active[l][s]) { hc.alpha[l][s] /= msqrt(sum_alphasq); }
        }
      }
    }

  }  // MODE 0,1

  const std::string str0 = "gra::spin::InitTMatrix:: [" + gra::aux::Spin2XtoString(2 * J) + ""^" +
                           gra::aux::ParityToString(P) + " -> " + gra::aux::Spin2XtoString(2 * s1) +
                           ""^" + gra::aux::ParityToString(P1) + " " +
                           gra::aux::Spin2XtoString(2 * s2) + ""^" + gra::aux::ParityToString(P2) +
                           "]" + " (identical = " + std::string(identical ? "true" : "false") + ")";

  if (nonzero == 0) {
    const std::string str = ": Decay is impossible (spin-parity-statistics) [.P_conservation = " +
                            (hc.P_conservation ? std::string("true") : std::string("false"));
    throw std::invalid_argument(str0 + str);
  }
  std::cout << std::endl;
  std::cout << "T matrix (2s1 + 1) x (2s2 + 1):" << std::endl;
  gra::matoper::PrintMatrixSeparate(hc.T);

  // Nullify all inactive
  for (std::size_t l = 0; l < hc.alpha.size_row(); ++l) {
    for (std::size_t s = 0; s < hc.alpha.size_col(); ++s) {
      if (active[l][s] == false) { hc.alpha[l][s] = 0.0; }
    }
  }

  std::cout << "Normalization:" << std::endl;
  std::cout << "sum_ls |alpha_{ls}|^2 = " << hc.alpha.FrobNorm2()
            << "  <=> sum_ls |T_{lambda1,lambda2}|^2 = " << hc.T.FrobNorm2() << std::endl
            << std::endl;

  // Check do have missing values:
  if (tag_set) {
    std::string str = "";
    for (std::size_t l = 0; l < need_to_set.size_row(); ++l) {
      for (std::size_t s = 0; s < need_to_set.size_col(); ++s) {
        if (need_to_set[l][s] == true) {
          str += "(" + std::to_string(l) + "," + std::to_string(s) + ") ";
        }
      }
    }

    const std::string middle =
        ", Following helicity decay alpha_ls-couplings "
```



```
        "missing from BRANCHING: (l,s) "
        "= ";
    throw std::invalid_argument(str0 + middle + str);
  }
}

// |l-s| <= J <= l + s, parity P = (-1)^l
// l is orbital angular momentum
// s is total spin
//
// Bose-Einstein statistics requires l-s to be even
// for identical Boson pairs.
bool BoseSymmetry(int l, int s) {
  const int number = l - s;
  if (number % 2 == 0) { return true; }
  return false;
}

// Fermi-Dirac statistics requires l+s to be even
// for identical Fermion pairs.
bool FermiSymmetry(int l, int s) {
  const int number = l + s;
  if (number % 2 == 0) { return true; }
  return false;
}

// Initial state spin treatment: Pomeron + Pomeron/Gamma -> Resonance X
//
//
std::complex<double> ProdAmp(const gra::LORENTZSCALAR &lts, gra::PARAM_RES &res) {
  // Apply coupling here
  std::complex<double> A0 = res.g;

  // Generation 2->1 spin correlations not active
  if (res.SPINGEN == false) { return A0; }

  // ------------------------------------------------------------------
  // Tensors J^P = 2+, 4+, 6+,...
  // "Pomeron effective spins"
  double s1 = 0;
  double s2 = 0;

  // SU(2) decomposition as given by InitTMatrix
  //
  // only one alpha_(ls) = 1.0 needed
  MMatrix<std::complex<double>> T0 = {{std::complex<double>(1.0)}};

  // ------------------------------------------------------------------

  // Scalar
  if (res.p.spinX2 == 0 && res.p.P == 1) { return A0; }

  // Pseudoscalar J^P = 0-, [e.g. eta(548), eta(958)^]
  if (res.p.spinX2 == 0 && res.p.P == -1) {
    /*
    s1 = 1;
    s2 = 1;
    // SU(2) decomposition as given by InitTMatrix
    //
    // only one alpha_(ls = 11) = 1.0
    //
    T0 = {{std::complex<double>(0.707), std::complex<double>(0.000), std::complex<double>( 0.000)},
          {std::complex<double>(0.000), std::complex<double>(0.000), std::complex<double>( 0.000)},
          {std::complex<double>(0.000), std::complex<double>(0.000), std::complex<double>(-0.707)}};
    */

    // This should be recursively connected with protons, instead use simplification below:

    // Forward proton pair deltaphi
    const double dphi = lts.pfinal[1].DeltaPhiAbs(lts.pfinal[2]);

    //  ^  WA102 data (not fully sin(phi) symmetric after MC, due to kinematics)
    //  |    .---.
    //  |  .     .
    //  |  .     .
    //  ------------->
    //  0          180 deg
    //
    // Two cases m1=m2=-1 and m1=m2=-1
    const double m1 = 1;
    const double m2 = 1;
```



```
    // See Kaidalov et al.
    auto ReggeTheory = [](double t1, double t2, double m1, double m2) {
      return std::pow(std::abs(t1), std::abs(m1) / 2.0) *
             std::pow(std::abs(t2), std::abs(m2) / 2.0);
    };

    return A0 * ReggeTheory(lts.t1, lts.t2, m1, m2) * std::sin(dphi);
  }

  // Axial vector J^P = 1+ [e.g. f1_1420]
  if (res.p.spinX2 == 2 && res.p.P == 1) {
    s1 = 1;
    s2 = 1;
    // SU(2) decomposition as given by InitTMatrix
    // only one alpha_(ls = 22) = 1
    //
    T0 = {{std::complex<double>(0.000), std::complex<double>(-0.500), std::complex<double>(0.000)},
          {std::complex<double>(0.500), std::complex<double>(0.000), std::complex<double>(-0.500)},
          {std::complex<double>(0.000), std::complex<double>(0.500), std::complex<double>(0.000)}};
  }

  // Vector photoproduction J^P = 1- [e.g. rho770]
  if (res.p.spinX2 == 2 && res.p.P == -1) {
    s1 = 0;
    s2 = 1;
    // SU(2) decomposition as given by InitTMatrix
    // assuming alpha_(ls = 01) = 1/sqrt(2)
    //          alpha_(ls = 21) = 1/sqrt(2)
    T0 = {
        {std::complex<double>(-0.697), std::complex<double>(-0.169), std::complex<double>(0.697)}};
  }

  // Boost propagator to the system rest frame
  M4Vec q1 = lts.q1;
  gra::kinematics::LorentzBoost(lts.pfinal[0], lts.pfinal[0].M(), q1, -1);

  const MMatrix<std::complex<double>> f0 =
      spin::fMatrix(T0, res.p.spinX2 / 2.0, s1, s2, q1.Theta(), q1.Phi());

  // ------------------------------------------------------------------
  // Construct the D-matrix for an event-by-event spin space density operator
  // rotation
  double theta_R = 0.0;
  double phi_R   = 0.0;

  // Fix the frame here
  const std::string FRAME = res.FRAME;
  GetRhoRotation(lts, FRAME, theta_R, phi_R);

  // ------------------------------------------------------------------
  // Rotation does mixing of spin states. N.B. Eigenvalues do not change in
  // rotation.
  const MMatrix<std::complex<double>> D = gra::spin::DMatrix(res.p.spinX2 / 2.0, theta_R, phi_R);

  // rho_rot = D^\dagger*rho*D [keep this sandwich order!]
  const MMatrix<std::complex<double>> rho_ROT = D.Dagger() * res.rho * D;

  // Krauss operator map
  const MMatrix<std::complex<double>> f0Rfd0 = f0 * rho_ROT * f0.Dagger();

  // Normalization as in DecayAmp
  const double NORM = msqrt(res.p.spinX2 + 1.0);

  // Apply final amplitude level weight
  A0 *= NORM * msqrt(std::real(f0Rfd0.Trace()));

  return A0;
}


// Recursive decay chain with spin correlations:
// X -> A > (A1 + A2) B > (B1 > (C1 + C2) + B2) ...
//
std::complex<double> DecayAmp(gra::LORENTZSCALAR &lts, gra::PARAM_RES &res) {
  // Apply coupling first
  double A0 = res.hel.g_decay;

  // Decay 1->2 spin correlations not active
  if (res.SPINDEC == false) { return A0; }

  // This function handles only 2-body decays
  if (lts.decaytree.size() != 2) { return A0; }
  // No spin, no need
```

```
416:   if (res.p.spinX2 == 0) {
417:     return res.hel.T[0][0] * A0;  // (1,s) = (0,0)
418:   }
419:
420:   // Transition amplitude matrix f for X -> A + B in the X rest frame
421:   // Daughter spins
422:   const double s1 = lts.decaytree[0].p.spinX2 / 2.0;
423:   const double s2 = lts.decaytree[1].p.spinX2 / 2.0;
424:
425:   // Boost daughter to the A-rest frame
426:   M4Vec boosted_daughter = lts.decaytree[0].p4;
427:   gra::kinematics::LorentzBoost(lts.pfinal[0], lts.pfinal[0].M(), boosted_daughter, -1);
428:
429:   MMatrix<std::complex<double>> fX = fMatrix(res.hel.T, res.p.spinX2 / 2.0, s1, s2,
430:                                              boosted_daughter.Theta(), boosted_daughter.Phi());
431:
432:   // Now go through the decay tree leaves recursively
433:   TreeRecursion(lts.decaytree[0]);
434:   TreeRecursion(lts.decaytree[1]);
435:
436:   // Get tensor products back
437:   MMatrix<std::complex<double>> fA;
438:   TensorTree(lts.decaytree[0], fA);
439:
440:   MMatrix<std::complex<double>> fB;
441:   TensorTree(lts.decaytree[1], fB);
442:
443:   // f matrices have dimensions: [(2s1 + 1)x(2s2 + 1)] x (2J + 1)
444:
445:   // Total transition amplitude matrix as a tensor product
446:   MMatrix<std::complex<double>> f;
447:   if (lts.decaytree[0].legs.size() == 0 && lts.decaytree[1].legs.size() == 0) {
448:     f = fX;  // No recursion
449:   } else {
450:     f = gra::matoper::TensorProd(fA, fB) * fX;  // Matrix x Matrix product
451:   }
452:
453:   MMatrix<std::complex<double>> fA_D = fA.Dagger();
454:
455:   // ------------------------------------------------------------------
456:   // Construct the D-matrix for an event-by-event spin space density operator
457:   // rotation
458:   double theta_R = 0.0;
459:   double phi_R   = 0.0;
460:   GetRhoRotation(lts, res.FRAME, theta_R, phi_R);
461:
462:   // ------------------------------------------------------------------
463:   // Rotation does mixing of spin states. N.B. Eigenvalues do not change in
464:   // rotation.
465:   const MMatrix<std::complex<double>> D = gra::spin::DMatrix(res.p.spinX2 / 2.0, theta_R, phi_R);
466:
467:   // rho_rot = D^\dagger*rho*D [keep this sandwich order!]
468:   const MMatrix<std::complex<double>> rho_ROT = D.Dagger() * res.rho * D;
469:
470:   // Weight (amplitude squared) of the event by the density matrix formalism:
471:   // Tr{f*rho*f^dagger}
472:   const MMatrix<std::complex<double>> fRfd = f * rho_ROT * f.Dagger();
473:
474:   // (2J+1) is normalization to match production coupling with spin-0
475:   // (same cross section obtained as with spin-0 if density
476:   //  matrix == I/(2J+1) == totally unpolarized)
477:   const double NORM = msqrt(res.p.spinX2 + 1.0);
478:
479:   // Final weight at amplitude level
480:   A0 *= NORM * msqrt(std::real(fRfd.Trace()));
481:
482:   return A0;
483: }
484:
485:
486: // Decay amplitude matrix X -> A + B
487: //
488: MMatrix<std::complex<double>> CalculateFMatrix(const MDecayBranch &branch) {
489:   const unsigned int A = 0;
490:   const unsigned int B = 1;
491:
492:   // Get daughter spins
493:   const double s1 = branch.legs[A].p.spinX2 / 2.0;
494:   const double s2 = branch.legs[B].p.spinX2 / 2.0;
495:
496:   // Rotate and boost daughters to the frame where z-spanned by X-flight direction (X-helicity rest
497:   // frame)
498:   std::vector<M4Vec> daughter = (branch.legs[A].p4, branch.legs[B].p4);
```



```
499:   gra::kinematics::HXframe(daughter, branch.p4);
500:
501:   return fMatrix(branch.hel.T, branch.p.spinX2 / 2.0, s1, s2, daughter[A].Theta(),
502:                  daughter[A].Phi());
503: }
504:
505: // Forward traverse the decay tree
506: //
507: //
508: void TreeRecursion(MDecayBranch &branch) {
509:   if (branch.legs.size() == 2) {
510:     branch.f = CalculateFMatrix(branch);
511:     // Infinite recursion
512:     for (const auto &i : aux::indices(branch.legs)) { TreeRecursion(branch.legs[i]); }
513:   }
514: }
515:
516:
517: // Reverse traverse the sequential decay tree
518: //
519: //
520: void TensorTree(const MDecayBranch &branch, MMatrix<std::complex<double>> &out) {
521:   if (branch.legs.size() == 2) {
522:     MMatrix<std::complex<double>> f0;
523:     MMatrix<std::complex<double>> f1;
524:
525:     TensorTree(branch.legs[0], f0);
526:     TensorTree(branch.legs[1], f1);
527:
528:     // Now we are traversing backwards the binary tree at branch points
529:     if (branch.legs[0].legs.size() == 0 && branch.legs[1].legs.size() == 0) {
530:       out = branch.f;
531:     } else {
532:       out = matoper::TensorProd(f0, f1) * branch.f;
533:     }
534:
535:     // Last leg unit vector
536:   } else {
537:     out = MMatrix<std::complex<double>>(1, branch.p.spinX2 + 1, 1.0);
538:   }
539: }
540:
541:
542: // Rotation angles to rotate the density matrix
543: //
544: void GetRhoRotation(const gra::LORENTZSCALAR &lts, const std::string &FRAME, double &theta_R,
545:                     double &phi_R) {
546:   // Spin polarization density matrix defined:
547:   // In direct non-rotated rest frame
548:   if (FRAME == "CM") {
549:     theta_R = 0;
550:     phi_R   = 0;
551:
552:     // In Helicity rest frame [quantization axis by system orientation in the lab]
553:   } else if (FRAME == "HX") {
554:     theta_R = lts.pfinal[0].Theta();  // +
555:     phi_R   = lts.pfinal[0].Phi();    // +
556:
557:
558:     // In Collins-Soper rest frame [quantization axis by the beam bijector frame]
559:   } else if (FRAME == "CS") {
560:     M4Vec p1b = lts.pbeam1;
561:     M4Vec p2b = lts.pbeam2;
562:
563:     // Boost the beam particles
564:     gra::kinematics::LorentzBoost(lts.pfinal[0], lts.pfinal[0].M(), p1b,
565:                                   -1);  // Note the minus sign
566:     gra::kinematics::LorentzBoost(lts.pfinal[0], lts.pfinal[0].M(), p2b,
567:                                   -1);  // Note the minus sign
568:
569:     // Now get the 3-momentum
570:     const std::vector<double> pb1boost3 = p1b.P3();
571:     const std::vector<double> pb2boost3 = p2b.P3();
572:
573:     // Collins-Soper bisector vector
574:     const std::vector<double> zaxis = gra::matoper::Unit(
575:         gra::matoper::Minus(gra::matoper::Unit(pb1boost3), gra::matoper::Unit(pb2boost3)));
576:
577:     M4Vec bijector;
578:     bijector.SetP3(zaxis);
579:
580:     // Now get the rotation angles
581:     theta_R = bijector.Theta();  // +
```



```
582:     phi_R   = bijector.Phi();    // +
583:
584:     // In Gottfried-Jackson frame [quantization axis by the momentum transfer vector]
585:   } else if (FRAME == "GJ") {
586:     // Propagator
587:     M4Vec qlboost = lts.ql;
588:     gra::kinematics::LorentzBoost(lts.pfinal[0], lts.pfinal[0].M(), qlboost,
589:                                   -1);  // Note the minus sign
590:
591:     theta_R = qlboost.Theta();  // +
592:     phi_R   = qlboost.Phi();    // +
593:
594:   } else {
595:     // Throw exception
596:     const std::string str =
597:         "gra::spin::GetRotation: Unknown polarization Lorentz rest FRAME chosen: " + FRAME +
598:         "[valid currently are: CM (no rotation), HX (helicity), CS (Collins-Soper)";
599:     throw std::invalid_argument(str);
600:   }
601: }
602:
603:
604: // Create spin projections -J <= M <= J (number of 2J+1)
605: // negative to positive
606: //
607: // Spin-1/2:  [-1/2, 1/2]
608: // Spin-1:    [-1, 0, 1]
609: // Spin-3/2:  [-3/2, -1/2, 1/2, 3/2]
610: // Spin-2:    [-2, -1, 0, 1, 2]
611: //
612: std::vector<double> SpinProjections(double J) {
613:   std::vector<double> M(static_cast<int>(2 * J + 1), 0.0);
614:   double              m = -J;
615:   for (std::size_t i = 0; i < M.size(); ++i) {
616:     M[i] += m;
617:     m += 1.0;
618:   }
619:   return M;
620: }
621:
622: // Returns transition amplitude matrix of size [(2s1 + 1)x(2s2 + 1)] x (2J + 1)
623: //
624: // f_{\lambda_1\lambda_2,M}(\theta,\phi) = D_{\lambda M}^J (\theta,\phi)
625: // T_{\lambda_1, \lambda_2}
626: //
627: // theta,phi defined in the decay rest frame
628: //
629: // [REFERENCE: Jacob, Wick, On the general theory of particles with spin, 1959]
630: // [REFERENCE: Amsler, Biot, Simulation of angular distributions, 1983]
631: MMatrix<std::complex<double>> fMatrix(const MMatrix<std::complex<double>> &T, double J, double s1,
632:                                       double s2, double theta, double phi) {
633:   if (T.size_row() != static_cast<unsigned int>(2 * s1 + 1) ||
634:       T.size_col() != static_cast<unsigned int>(2 * s2 + 1)) {
635:     throw std::invalid_argument(
636:         "gra::spin::fMatrix: Input T matrix "
637:         "[2s1+1)x(2s2+1) with wrong dimensions: " +
638:         std::to_string(T.size_row()) + " x " + std::to_string(T.size_col()));
639:   }
640:   // Number of final state configurations
641:   const unsigned int N =
642:       static_cast<unsigned int>(2 * s1 + 1) * static_cast<unsigned int>(2 * s2 + 1);
643:
644:   // Initial state projections
645:   const std::vector<double> M = SpinProjections(J);
646:
647:   // ------------------------------------------------------------------
648:   // Construct final state helicity configurations
649:
650:   // For indexing T matrix
651:   MMatrix<double>       lambda_values(N, 2, 0.0);
652:   MMatrix<unsigned int> lambda_index(N, 2, 0);
653:
654:   unsigned int i = 0;
655:   for (int mu = 0; mu < static_cast<int>(2 * s1 + 1); ++mu) {
656:     for (int nu = 0; nu < static_cast<int>(2 * s2 + 1); ++nu) {
657:       lambda_values[i][0] = -s1 + static_cast<double>(mu);
658:       lambda_values[i][1] = -s2 + static_cast<double>(nu);
659:
660:       lambda_index[i][0] = mu;
661:       lambda_index[i][1] = nu;
662:       ++i;
663:     }
664:   }
```



```
665:
666:   // ------------------------------------------------------------------
667:   // Construct transition amplitude matrix
668:   MMatrix<std::complex<double>> f(N, 2 * J + 1, 0.0);
669:
670:   // Rows = final state spin projections
671:   for (std::size_t i = 0; i < N; ++i) {
672:     // Final state helicities
673:     const double lambda1 = lambda_values[i][0];
674:     const double lambda2 = lambda_values[i][1];
675:
676:     // Total lambda by definition
677:     const double lambda = lambda1 - lambda2;
678:
679:     // Columns = initial state polarizations
680:     for (std::size_t j = 0; j < 2 * J + 1; ++j) {
681:       // Wigner D * Helicity amplitude
682:       f[i][j] = WignerD(theta, phi, lambda, M[j], J) * T[lambda_index[i][0]][lambda_index[i][1]];
683:     }
684:   }
685:   // std::cout << "f::" << std::endl;
686:   // gra::matoper::PrintMatrixSeparate(f);
687:   // exit(1);
688:
689:   return f;
690: }
691:
692: // Construct Wigner D-matrix for a spin-space operator rotation
693: MMatrix<std::complex<double>> DMatrix(double J, double theta_R, double phi_R) {
694:   // Create spin projections
695:   const std::vector<double> M = SpinProjections(J);
696:
697:   const unsigned int n = static_cast<unsigned int>(2 * J + 1);
698:
699:   MMatrix<std::complex<double>> D(n, n, 0.0);
700:   for (std::size_t i = 0; i < n; ++i) {
701:     for (std::size_t j = 0; j < n; ++j) { D[i][j] = WignerD(theta_R, phi_R, M[i], M[j], J); }
702:   }
703:   return D;
704: }
705:
706: // SU(2) Clebsch-Gordan coefficients
707: double ClebschGordan(double j1, double j2, double m1, double m2, double j, double m) {
708:   // Selection rules
709:   if (!CGrules(j1, j2, m1, m2, j, m)) { return 0.0; }
710:
711:   // Use Wigner-3j symbol to evaluate, note minus on m!
712:   return std::pow(-1.0, m + j1 - j2) * msqrt(2 * j + 1) * W3j(j1, j2, j, m1, m2, -m);
713: }
714:
715:
716: // Check if half-integer
717: bool chalfint(double x) {
718:   if (std::abs(2 * x - std::floor(2 * x)) < 1E-9) { return true; }
719:   return false;
720: }
721:
722: // Check if integer
723: bool cint(double x) {
724:   if (std::abs(x - std::floor(x)) < 1E-9) { return true; }
725:   return false;
726: }
727:
728: // Floating point equality comparison
729: bool isequal(double x, double y) {
730:   if (std::abs(x - y) < 1E-9) { return true; }
731:   return false;
732: }
733:
734: // CG-selection rules
735: double CGrules(double j1, double j2, double m1, double m2, double j, double m) {
736:   const bool PRINT_ON = false;
737:
738:   if (!chalfint(j1) || !chalfint(j2) || !chalfint(j)) {
739:     if (PRINT_ON) printf("CGrules:: (j1,j2,j) not integer or half-integer\n");
740:
741:     return false;
742:   }
743:   if (!cint(j1 + j2 + j)) {  // "Integer perimeter rule"
744:     if (PRINT_ON) printf("CGrules:: j1 + j2 + j not integer\n");
745:
746:     return false;
747:   }
```

```
748:   if (!chalfint(m1) || !chalfint(m2) || !chalfint(m)) {
749:     if (PRINT_ON) printf("CGrules:: (m1,m2,m) not integer or half-integer\n");
750:
751:     return false;
752:   }
753:   if (!isequal(m1 + m2, m)) {
754:     if (PRINT_ON) printf("CGrules:: m1 + m2 != m\n");
755:
756:     return false;
757:   }
758:   if (!isequal(j1 - m1, std::floor(j1 - m1))) {
759:     if (PRINT_ON) printf("CGrules:: parity of 2j_1 and 2m_1 does not match\n");
760:
761:     return false;
762:   }
763:   if (!isequal(j2 - m2, std::floor(j2 - m2))) {
764:     if (PRINT_ON) printf("CGrules:: parity of 2j_2 and 2m_2 does not match\n");
765:
766:     return false;
767:   }
768:   if (!isequal(j - m, std::floor(j - m))) {
769:     if (PRINT_ON) printf("CGrules:: parity of 2j and 2m does not match\n");
770:
771:     return false;
772:   }
773:   if ((j1 + j2) < j || j < std::abs(j1 - j2)) {
774:     if (PRINT_ON) printf("CGrules:: j over valid domain\n");
775:
776:     return false;
777:   }
778:   if (std::abs(m1) > j1) {
779:     if (PRINT_ON) printf("CGrules:: |m1| > j1\n");
780:
781:     return false;
782:   }
783:   if (std::abs(m2) > j2) {
784:     if (PRINT_ON) printf("CGrules:: |m2| > j2\n");
785:
786:     return false;
787:   }
788:   if (std::abs(m) > j) {
789:     if (PRINT_ON) printf("CGrules:: |m| > j\n");
790:
791:     return false;
792:   }
793:   return true;
794: }
795:
796: // Wigner 3j symbol
797: //
798: // (j1 j2 j3)
799: // (m1 m1 m3)
800: //
801: // http://mathworld.wolfram.com/Wigner3j-Symbol.html
802:
803: double W3j(double j1, double j2, double j3, double m1, double m2, double m3) {
804:   // Check selection rules
805:   if (!W3jrules(j1, j2, j3, m1, m2, m3)) { return 0.0; }
806:
807:   // Create coefficient structure
808:   const int c1 = -j2 + m1 + j3;
809:   const int c2 = -j1 - m2 + j3;
810:   const int c3 = j1 + j2 - j3;
811:   const int c4 = j1 - m1;
812:   const int c5 = j2 + m2;
813:
814:   // Boundaries for which factorials are non-negative
815:   const int lower = std::max(std::max(0, -c1), -c2);
816:   const int upper = std::min(std::min(c3, c4), c5);
817:   double    fsum  = 0;
818:
819:   // Evaluate sum term
820:   for (int k = lower; k <= upper; ++k) {
821:     fsum += std::pow(-1.0, k) / (gra::math::factorial(k) * gra::math::factorial(c1 + k) *
822:                                  gra::math::factorial(c2 + k) * gra::math::factorial(c3 - k) *
823:                                  gra::math::factorial(c4 - k) * gra::math::factorial(c5 - k));
824:   }
825:
826:   // Evaluate by Racah formula
827:   const double wc = std::pow(-1.0, j1 - j2 - m3) *
828:                     msqrt(TriangleCoeff(j1, j2, j3) * gra::math::factorial(j1 + m1) *
829:                           gra::math::factorial(j1 - m1) * gra::math::factorial(j2 + m2) *
830:                           gra::math::factorial(j2 - m2) * gra::math::factorial(j3 + m3) *
```



```
831:                           gra::math::factorial(j3 - m3)) *
832:                     fsum;
833:   return wc;
834: }
835:
836: // Triangle coefficient
837: double TriangleCoeff(double j1, double j2, double j3) {
838:   return gra::math::factorial(j1 + j2 - j3) * gra::math::factorial(j1 - j2 + j3) *
839:          gra::math::factorial(-j1 + j2 + j3) / gra::math::factorial(j1 + j2 + j3 + 1);
840: }
841:
842: // Selection rules
843: bool W3jrules(double j1, double j2, double j3, double m1, double m2, double m3) {
844:   const bool PRINT_ON = false;
845:
846:   if (!cint(j1 + j2 + j3)) {
847:     if (PRINT_ON) { std::cout << "W3jSR:: j1 + j2 + j3 not integer\n"; }
848:
849:     return false;
850:   }
851:   if (!isequal(m1 + m2 + m3, 0)) {
852:     if (PRINT_ON) { std::cout << "W3jSR:: m1 + m2 + m3 != 0\n"; }
853:
854:     return false;
855:   }
856:   if (!isequal(j1 - m1, std::floor(j1 - m1))) {
857:     if (PRINT_ON) { std::cout << "W3jSR:: parity of 2j_1 and 2m_1 does not match\n"; }
858:
859:     return false;
860:   }
861:   if (!isequal(j2 - m2, std::floor(j2 - m2))) {
862:     if (PRINT_ON) { std::cout << "W3jSR:: parity of 2j_2 and 2m_2 does not match\n"; }
863:
864:     return false;
865:   }
866:   if (!isequal(j3 - m3, std::floor(j3 - m3))) {
867:     if (PRINT_ON) { std::cout << "W3jSR:: parity of 2j_3 and 2m_3 does not match\n"; }
868:
869:     return false;
870:   }
871:   if ((j1 + j2) < j3 || j3 < std::abs(j1 - j2)) {
872:     if (PRINT_ON) { std::cout << "W3jSR:: j over valid domain\n"; }
873:
874:     return false;
875:   }
876:   if (std::abs(m1) > j1) {
877:     if (PRINT_ON) { std::cout << "W3jSR:: |m1| > j1\n"; }
878:
879:     return false;
880:   }
881:   if (std::abs(m2) > j2) {
882:     if (PRINT_ON) { std::cout << "W3jSR:: |m2| > j2\n"; }
883:
884:     return false;
885:   }
886:   if (std::abs(m3) > j3) {
887:     if (PRINT_ON) { std::cout << "W3jSR:: |m3| > j3\n"; }
888:
889:     return false;
890:   }
891:   return true;
892: }
893:
894: // Calculate quantum mechanical von Neumann entropy of a density matrix
895: double VonNeumannEntropy(const MMatrix<std::complex<double>> &rho) {
896:   const unsigned int N = rho.size_col();
897:   Eigen::MatrixXcd   A(N, N);
898:
899:   // Collect matrix elements
900:   for (std::size_t i = 0; i < N; ++i) {
901:     for (std::size_t j = 0; j < N; ++j) { A(i, j) = rho[i][j]; }
902:   }
903:
904:   // Solve eigenvalues
905:   Eigen::SelfAdjointEigenSolver<Eigen::MatrixXcd> es(A);
906:   if (es.info() != Eigen::Success) { return -1.0; }
907:
908:   // Calculate von Neumann Entropy
909:   double entropy = 0.0;
910:   for (std::size_t i = 0; i < N; ++i) {
911:     std::complex<double> lambda = es.eigenvalues().col(0)[i];
912:     const double         eigval = std::real(lambda);  // Take real part for numerics
913:     if (eigval > epsilon) { entropy += eigval * std::log(eigval); }
```



```
914:   }
915:   entropy = -entropy;
916:   return entropy;
917: }
918:
919: // Wigner small-d (real valued convention, z-y-z convention)
920: //
921: // Wigner small d_{mm'}^J(theta) ~ rotation around y_z-axis
922: //
923: // [REFERENCE: en.wikipedia.org/wiki/Wigner_D-matrix#Wigner_{small}_d-matrix]
924: //
925: // m, mp, J all in integers (not half-integers)
926: //
927: double WignerSmalld(double theta, double m, double mp, double J) {
928:   const double factor =
929:       gra::math::msqrt(gra::math::factorial(J + m) * gra::math::factorial(J - m) *
930:                        gra::math::factorial(J + mp) * gra::math::factorial(J - mp));
931:
932:   // k = 0,1,...,2J
933:   for (int k = 0; k <= 2 * J; ++k) {
934:     double value = 0;
935:     double val1  = 0.0;
936:     double val2  = 0.0;
937:
938:     // Ignore terms with negative factorials
939:     if (J - mp - k >= 0 && J + m - k >= 0 && k + mp - m >= 0 && k >= 0) {
940:       val1 = std::pow(-1, mp - m + k) /
941:              (gra::math::factorial(J - mp - k) * gra::math::factorial(J + m - k) *
942:               gra::math::factorial(k + mp - m) * gra::math::factorial(k));
943:
944:       val2 = std::pow(std::cos(theta / 2.0), 2 * J + m - mp - 2 * k) *
945:              std::pow(std::sin(theta / 2.0), mp - m + 2 * k);
946:
947:       value = val1 * val2;
948:     }
949:     s += value;
950:   }
951:   // Fatal error
952:   if (std::isnan(s)) {
953:     std::string str =
954:         "gra::spin::WignerSmalld: NaN value with (theta,m,mp,J) = " + std::to_string(theta) + " " +
955:         std::to_string(m) + " " + std::to_string(mp) + " " + std::to_string(J);
956:     throw std::invalid_argument(str);
957:   }
958:   return factor * s;
959: }
960:
961:
962: // Wigner large-D
963: // Wigner large D(theta,phi) = exp(iJ_y theta) exp(iJ_z phi)
964: // D = @(theta,phi,m,mp,J) wignersmalld(theta,m,mp,J)*exp(1i*mp*phi);
965: //
966: // m, mp, J all in integers (not half-integers)
967: //
968: std::complex<double> WignerD(double theta, double phi, double m, double mp, double J) {
969:   return WignerSmalld(theta, m, mp, J) * std::exp(std::complex<double>(0, 1) * mp * phi);
970: }
971:
972: // Density matrix properties:
973: //
974: //  1. rho^2 = rho         (projector)
975: //  2. rho^\dagger = rho   (hermiticity)
976: //  3. Tr[rho] = 1         (normalization)
977: //  4. rho >= 0            (positivity, eigenvalues greater or equal to zero)
978: //
979: // Test positivity of the density matrix,
980: // - Also checks the trivial fact that rho
981: //   has right dimensions given J, that is (2J+1)
982: // - Also checks that the normalization is ok.
983: //
984: //
985: // TBD, write down generalization for any spin (e.g. done with Eigen library
986: // eigenvalues)
987:
988: bool Positivity(const MMatrix<std::complex<double>> &rho, unsigned int J) {
989:   const unsigned int n = rho.size_row();
990:
991:   // Dimensions
992:   if ((n - 1) / 2 != J) {
993:     std::string str =
994:         "gra::spin::Positivity: Density matrix does not have a valid "
995:         "dimension (2J+1) !";
996:     throw std::invalid_argument(str);
```



```
997:     return false;
998:   }
999:
1000:   // Normalization, abs taken for comparison
1001:   const double tracerho = std::abs(rho.Trace());
1002:   if (std::abs(tracerho - 1.0) > 1e-2) {
1003:     std::string str =
1004:         "gra::spin::Positivity: Density matrix is not properly "
1005:         "normalized (Tr[rho] = " +
1006:         std::to_string(tracerho) + ") (should be 1) !";
1007:     gra::matoper::PrintMatrixSeparate(rho);
1008:     throw std::invalid_argument(str);
1009:     return false;
1010:   }
1011:
1012:   const double NEG_EPS = -1e6;  // Negative for floating point reasons
1013:
1014:   // Spin-1
1015:   if (J == 1) {
1016:     // Component as:
1017:     //
1018:     //     rho_i = [0.5*(1-a), b+1i*c, d;
1019:     //              b-1i*c,    a, -b+1i*c;
1020:     //              d, -b-1i*c, 0.5*(1-a)];
1021:
1022:     // Test positivity of rho (ANALYTICAL)
1023:     const double a = std::real(rho[1][1]);
1024:     const double b = std::real(rho[0][1]);
1025:     const double c = std::imag(rho[0][1]);
1026:     const double d = std::real(rho[0][2]);
1027:
1028:     const double pc1 = -4 * b * b - 4 * c * c - 2 * d * d + a - (3 * a * a) / 2.0 + 1 / 2.0;
1029:     const double pc2 =
1030:         -(3 * (2 * d - a + 1) * (2 * a * d - a + a * a + 4 * b * b + 4 * c * c)) / 2.0;
1031:
1032:     if (pc1 >= NEG_EPS && pc2 >= NEG_EPS) {
1033:       return true;
1034:     } else {
1035:       printf("pc1 = %0.5f, pc2 = %0.5f \n", pc1, pc2);
1036:       std::string str =
1037:           "gra::spin::Positivity: Eigenvalues of the density "
1038:           "matrix are not >= 0 !";
1039:       gra::matoper::PrintMatrixSeparate(rho);
1040:       throw std::invalid_argument(str);
1041:       return false;
1042:     }
1043:   }
1044:
1045:   // Spin-2
1046:   else if (J == 2) {
1047:     // Components as:
1048:     //
1049:     //     rho_i  = [a,       f+1i*c,   g+1i*d,      h+1i*e,      m;
1050:     //               f-1i*c,  b,        j+1i*k,      1, -h+1i*e;
1051:     //               g-1i*d,  j-1i*k,   1-2*(a+b),  -j+1i*k, g-1i*d;
1052:     //               h-1i*e,  1,        -j-1i*k,     b, -f+1i*c;
1053:     //               m,       -h-1i*e,  g+1i*d,      -f-1i*c, a];
1054:
1055:     // Test positivity of rho (ANALYTICAL)
1056:     const double a = std::real(rho[0][0]);
1057:     const double b = std::real(rho[1][1]);
1058:     const double c = std::imag(rho[0][1]);
1059:     const double d = std::imag(rho[0][2]);
1060:     const double e = std::imag(rho[0][3]);
1061:     const double f = std::real(rho[0][1]);
1062:     const double g = std::real(rho[0][2]);
1063:     const double h = std::real(rho[0][3]);
1064:     const double j = std::real(rho[1][2]);
1065:     const double k = std::imag(rho[1][2]);
1066:     const double l = std::real(rho[1][3]);
1067:     const double m = std::real(rho[0][4]);
1068:
1069:     const double pc1 = -6 * a * a - 8 * a * b + 4 * a - 6 * b * b + 4 * b - 4 * c * c - 4 * d * d -
1070:                        4 * e * e - 4 * f * f - 4 * g * g - 4 * h * h - 4 * j * j - 4 * k * k -
1071:                        2 * l * l - 2 * m * m;
1072:     const double pc2 = -12 * a * a * a - 48 * a * a * b + 6 * a * a - 48 * a * b * b + 24 * a * b +
1073:                        12 * a * c * c - 12 * a * d * d + 12 * a * e * e + 12 * a * f * f -
1074:                        12 * a * g * g + 12 * a * h * h - 24 * a * j * j - 24 * a * k * k +
1075:                        12 * a * m * m - 12 * b * b * b + 6 * b * b + 12 * b * c * c -
1076:                        24 * b * d * d + 12 * b * e * e + 12 * b * f * f - 24 * b * g * g +
1077:                        12 * b * h * h - 12 * b * j * j - 12 * b * k * k + 12 * b * l * l -
1078:                        12 * c * c * 24 * c * d * j + 24 * c * e * l - 24 * c * e * m -
1079:                        24 * c * g * k + 12 * d * d * m - 24 * d * e * j + 24 * d * f * k -
```

```
                        24 * d * h * k - 12 * e * e + 24 * e * g * k - 12 * f * f + 24 * f * g * j +
                        24 * f * h * l - 24 * f * h * m + 12 * g * g * m - 24 * g * h * j -
                        12 * h * h - 12 * j * j * l - 12 * k * k * l - 6 * l * l - 6 * m * m;
    const double pc3 =
          48 * a * a * b - 168 * a * a * b * b - 96 * a * a * a * b + 96 * a * a * c * c +
          96 * a * a * e * e + 96 * a * a * f * f + 96 * a * a * h * h - 48 * a * a * j * j -
          48 * a * a * k * k + 72 * a * a * l * l - 96 * a * b * b * b + 48 * a * b * b +
          144 * a * b * c * c - 96 * a * b * d * d + 144 * a * b * e * e + 144 * a * b * f * f -
          96 * a * b * g * g + 144 * a * b * h * h - 96 * a * b * j * j - 96 * a * b * k * k +
          96 * a * b * l * l + 96 * a * b * m * m - 48 * a * c * c + 96 * a * c * d * j -
          96 * a * c * e * l + 192 * a * c * e * m - 96 * a * c * g * k - 96 * a * d * e * j +
          96 * a * d * f * k - 96 * a * d * h * k - 48 * a * e * e + 96 * a * e * g * k -
          48 * a * f * f + 96 * a * f * g * j - 96 * a * f * h * l + 192 * a * f * h * m -
          96 * a * g * h * j - 48 * a * h * h - 96 * a * j * l - 96 * a * k * k * l -
          48 * a * l * l + 96 * b * b * c * c - 48 * b * b * d * d + 96 * b * b * e * e +
          96 * b * b * f * f - 48 * b * b * g * g + 96 * b * b * h * h + 72 * b * b * m * m -
          48 * b * c * c + 96 * b * c * d * j - 192 * b * c * e * l + 96 * b * c * e * m -
          96 * b * c * g * k + 96 * b * d * d * m - 96 * b * d * e * j + 96 * b * d * f * k -
          96 * b * d * h * k - 48 * b * e * e + 96 * b * e * g * k - 48 * b * f * f +
          96 * b * f * g * j - 192 * b * f * h * l + 96 * b * f * h * m + 96 * b * g * g * m -
          96 * b * g * h * j - 48 * b * h * h - 48 * b * m * m + 24 * c * c * c + c +
          48 * c * c * d * d - 48 * c * c * e * e + 48 * c * c * f * f + 48 * c * c * g * g +
          48 * c * c * h * h + 48 * c * c * j * j + 48 * c * c * k * k + 48 * c * c * l * m +
          96 * c * d * d * e + 96 * c * d * j * l - 96 * c * d * j * m - 192 * c * e * f * h +
          96 * c * e * g * g + 96 * c * e * j * j + 96 * c * e * k * k + 96 * c * e * l -
          96 * c * e * m - 96 * c * g * k * l + 96 * c * g * k * m + 48 * d * d * e * e +
          48 * d * d * f * f + 96 * d * d * f * h + 48 * d * d * h * h + 48 * d * d * l * l -
          96 * d * e * j * l + 96 * d * e * j * m + 96 * d * f * k * l - 96 * d * f * k * m -
          96 * d * h * k * l + 96 * d * h * k * m + 24 * e * e * e * e + 48 * e * e * f * f +
          48 * e * e * g * g + 48 * e * e * h * h + 48 * e * e * j * j + 48 * e * e * k * k +
          48 * e * e * l * m + 96 * e * g * k * l - 96 * e * g * k * m + 24 * f * f * f * f +
          48 * f * f * g * g - 48 * f * f * h * h + 48 * f * f * j * j + 48 * f * f * k * k +
          48 * f * f * l * m + 96 * f * g * g * h + 96 * f * g * j * l - 96 * f * g * j * m +
          96 * f * h * j * j + 96 * f * h * k * k + 96 * f * h * l - 96 * f * h * m +
          48 * g * g * h * h + 48 * g * g * l * l - 96 * g * h * j * l + 96 * g * h * j * m +
          24 * h * h * h * h + 48 * h * h * j * j + 48 * h * h * k * k + 48 * h * h * l * m +
          48 * j * j * m * m + 48 * k * k * m * m + 24 * l * l * l * m * m;
    const double pc4 =
          120 *
          (c * c + 2 * c * e + e * e + f * f + 2 * f * h + h * h - a * b - a * l + b * m + l * m) *
          (a * l - 2 * c * e - a * b - 2 * f * h - b * m + l * m + 2 * a * b * b + 2 * a * a * b -
           2 * a * c * c - 2 * b * c * c - 2 * a * e * e + 2 * b * d * d - 2 * a * f * f -
           2 * b * e * e - 2 * b * f * f - 2 * a * h * h + 2 * b * g * g - 2 * b * h * h +
           2 * a * j * j + 2 * a * k * k - 2 * a * a * l + 2 * b * b * m - 2 * d * d * l -
           2 * g * g * l + 2 * j * j * m + 2 * k * k * m + c * c + e * e + f * f + h * h +
           4 * a * c * e + 4 * b * c * e - 2 * a * b * l + 4 * a * f * h + 2 * a * b * m +
           4 * b * f * h - 4 * c * d * j + 4 * d * e * j + 4 * c * g * k - 4 * d * f * k +
           4 * d * h * k - 4 * e * g * k - 4 * f * g * j + 4 * g * h * j - 2 * a * l * m -
           2 * b * l * m);

    if (pc1 >= NEG_EPS && pc2 >= NEG_EPS && pc3 >= NEG_EPS && pc4 >= NEG_EPS) {
      return true;
    } else {
      printf(
          "pc1 = %0.5f, pc2 = %0.5f, pc3 = %0.5f, pc4 = "
          "%0.5f \n",
          pc1, pc2, pc3, pc4);
      std::string str =
          "gra::spin::Positivity: Eigenvalues of the density "
          "matrix are not >= 0 !";
      gra::matoper::PrintMatrixSeparate(rho);
      throw std::invalid_argument(str);
      return false;
    }
  } else {
    std::string str =
        "gra::spin::Positivity: Not a valid spin, only J = 1 and 2 currently tested -- check "
        "manually";
    gra::matoper::PrintMatrixSeparate(rho);
    return true;
  }
}

// Generate random density matrix for the resonance
// Create random (under Haar / Hilbert-Schmidt measure) spin-density matrices
// for any spin,
// and for spin 1 and 2 the possibility to enforce parity conservation
//
//
// TBD, write down the generalization of parity conservation constaint for any
// spin.
MMatrix<std::complex<double>> RandomRho(unsigned int J, bool parity, MRandom &rng) {
  const std::complex<double>    zi(0, 1);  // imag unit
```



```
  MMatrix<std::complex<double>> rho;

  // Draw random density matrices until valid found, takes a couple
  // usually
  while (true) {
    // Create random complex matrix (n x n)
    const unsigned int             n = 2 * J + 1;
    MMatrix<std::complex<double>> r(n, n);

    for (std::size_t i = 0; i < n; ++i) {
      for (std::size_t j = 0; j < n; ++j) {
        r[i][j] = rng.G(0, 1) + zi * rng.G(0, 1);  // real and imag part from
                                                   // normal distribution
      }
    }
    // Take product (outer product per vector): r*r^\dagger
    rho = r * r.Dagger();

    // Normalize the trace to 1 (real just for conversion to double)
    const double scale = 1.0 / std::real(rho.Trace());
    rho                = rho * scale;

    // If parity conservation required, then symmetrize
    if (parity) {
      if (J == 1) {
        // 1.1 Purely real elements by parity and
        // hermiticity
        rho[0][2] = std::real(rho[0][2]);

        // 1.2 Parity constrained
        rho[1][2] = -std::real(rho[0][1]) + zi * std::imag(rho[0][1]);

        // 2. Lower triangle = hermiticity constrained
        for (std::size_t i = 0; i < n; ++i) {
          for (std::size_t j = 0; j <= i; ++j) { rho[i][j] = std::conj(rho[j][i]); }
        }

        // 3. Diagonal parity constrained
        rho[2][2] = std::real(rho[0][0]);
      }
      if (J == 2) {
        // 1.1 Purely real elements by parity and
        // hermiticity
        rho[0][4] = std::real(rho[0][4]);
        rho[1][3] = std::real(rho[1][3]);

        // 1.2 Parity constrained
        rho[1][4] = -std::real(rho[0][3]) + zi * std::imag(rho[0][3]);
        rho[2][3] = -std::real(rho[1][2]) + zi * std::imag(rho[1][2]);
        rho[2][4] = std::conj(rho[0][2]);
        rho[3][4] = -std::real(rho[0][1]) + zi * std::imag(rho[0][1]);

        // 2. Lower triangle = hermiticity constrained
        for (std::size_t i = 0; i < n; ++i) {
          for (std::size_t j = 0; j <= i; ++j) { rho[i][j] = std::conj(rho[j][i]); }
        }

        // 3. Diagonal parity constrained
        rho[3][3] = std::real(rho[1][1]);
        rho[4][4] = std::real(rho[0][0]);
      }

      // Re-Normalize trace to 1 (real just for conversion to double)
      const double newscale = 1.0 / std::real(rho.Trace());
      rho                   = rho * newscale;
      ;
    }

    // printmatrix(rho);
    // Check that the matrix is valid density matrix in terms of
    // positivity requirement
    try {
      if (Positivity(rho, J)) { break; }
    } catch (...) {
      continue;  // not valid density matrix
    }
  }

  gra::matoper::PrintMatrixSeparate(rho);

  return rho;
}
```



```
}  // namespace spin
}  // namespace gra
```



```
// PDG class
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

#include <map>
#include <regex>

// Own
#include "Granitti/MPDG.h"

// Libraries
#include "rang.hpp"

namespace gra {

// Read PDG particle data in
// Input as the full path to PDG .mcd file
void MPDG::ReadParticleData(const std::string &filepath) {
  std::cout << "MPDG::ReadParticleData: Reading in PDG tables: ";
  std::ifstream infile(filepath);
  std::string   line;

  // Clear particle data
  PDG_table.clear();

  while (std::getline(infile, line)) {
    std::istringstream iss(line);

    // Check * (comment) lines
    std::string first;
    iss >> first;
    std::string str = first;
    str.erase(str.begin() + 1, str.end());

    // Skip lines with *
    if (str.compare("*") == 0) continue;

    //  ******************** DEFINITION FROM PDG ******************
    //       1 -  8 \ Monte Carlo particle numbers as described in
    //      the "Review of
    //       9 - 16 | Particle Physics". Charge states appear, as
    //      appropriate,
    //      17 - 24 | from left-to-right in the order -, 0, +, ++.
    //      25 - 32 /
    //          33   blank
    //      34 - 51   central value of the mass (double precision)
    //          52   blank
    //      53 - 60   positive error
    //          61   blank
    //      62 - 69   negative error
    //          70   blank
    //      71 - 88   central value of the width (double precision)
    //          89   blank
    //      90 - 97   positive error
    //          98   blank
    //      99 -106   negative error
    //         107   blank
    //     108 -128   particle name left-justified in the field and
    //                charge states right-justified in the field.
    //                This field is for ease of visual examination
    //                of the file and
    //                should not be taken as a standardized
    //                presentation of
    //                particle names.

    const int L         = 40;
    char      cid[L]    = {0};
    char      cmass[L]  = {0};
    char      cwidth[L] = {0};
    char      cname[L]  = {0};

    sscanf(line.c_str(), "%33c %35c %35c %21c", cid, cmass, cwidth, cname);
    // printf("%s\t%s\t%s\t%s \n", cid, cmass, cwidth, cname);

    std::string sid(cid, L);
    std::string smass(cmass, L);
    std::string swidth(cwidth, L);
    std::string sname(cname, L);

    std::string temp_str;

    // ------------------------------------------------------
```

```
84:     // ID
85:     // First find out how many charges
86:     std::vector<int>  id;
87:     std::istringstream ss(sid);
88:
89:     while (ss >> temp_str) {
90:       // Convert string to int
91:       int value = 0;
92:       std::stringstream(temp_str) >> value;
93:       if (value != 0) { id.push_back(value); }
94:     }
95:
96:     // -----------------------------------------------------
97:     // MASS
98:     // Mass and errors
99:     std::vector<double> mass;
100:    std::stringstream   ss2(smass);
101:
102:    while (ss2 >> temp_str) {
103:      // Convert string to int
104:      double value = 0;
105:      std::stringstream(temp_str) >> value;
106:      mass.push_back(value);
107:    }
108:
109:    // -----------------------------------------------------
110:    // WIDTH
111:    // Width and errors
112:    std::vector<double> width;
113:    std::stringstream   ss3(swidth);
114:
115:    // If the particle WIDTH columns were empty
116:    // -> name column string will be empty by reading logic. This
117:    // checks that.
118:    bool empty = true;
119:    if ((sname.find("0") != std::string::npos) || (sname.find("+") != std::string::npos) ||
120:        (sname.find("-") != std::string::npos) || (sname.find("++") != std::string::npos)) {
121:      empty = false;
122:    }
123:
124:    if (!empty) {
125:      while (ss3 >> temp_str) {
126:        // Convert string to double
127:        double value = 0;
128:        std::stringstream(temp_str) >> value;
129:        width.push_back(value);
130:      }
131:    }
132:
133:    // -----------------------------------------------------
134:    // NAME and Charges
135:    std::stringstream ss4;
136:    if (empty == true) {
137:      ss4 = std::stringstream(swidth);
138:    } else {
139:      ss4 = std::stringstream(sname);
140:    }
141:    std::string name;
142:    ss4 >> name;
143:
144:    // Split by comma
145:    std::vector<int> chargeX3;
146:
147:    while (ss4.good()) {
148:      std::string substr;
149:      std::getline(ss4, substr, ',');
150:
151:      // Remove extra whitespace
152:      substr = std::regex_replace(substr, std::regex("^ +| +$|( ) +"), "$1");
153:
154:      // Identify charge (ORDER is important here!)
155:      if (substr.find("++") != std::string::npos) {
156:        chargeX3.push_back(6);
157:      } else if (substr.find("--") != std::string::npos) {
158:        chargeX3.push_back(-6);
159:      } else if (substr.find("-1/3") != std::string::npos) {
160:        chargeX3.push_back(-1);
161:      } else if (substr.find("+2/3") != std::string::npos) {
162:        chargeX3.push_back(2);
163:      } else if (substr.find("+") != std::string::npos) {
164:        chargeX3.push_back(3);
165:      } else if (substr.find("-") != std::string::npos) {
166:        chargeX3.push_back(-3);
```



```
167:      } else if (substr.find("0") != std::string::npos) {
168:        chargeX3.push_back(0);
169:      }
170:    }
171:
172:    // Loop over different charge assignments
173:    for (std::size_t k = 0; k < id.size(); ++k) {
174:      // New particle
175:      gra::MParticle p;
176:
177:      // Add properties
178:      p.name = name;
179:
180:      p.pdg = id[k];
181:      if (mass.size() > 0) { p.mass = mass[0]; }
182:      if (width.size() > 0) {  // Unstable particles have width
183:        p.width = width[0];
184:      }
185:      p.chargeX3 = chargeX3[k];
186:      p.tau      = PDG::hbar / p.width;  // mean lifetime in the rest frame
187:
188:      int lastdigit = p.pdg % 10;       // Get last digit
189:      p.spinX2      = (lastdigit - 1);  // Get 2J
190:      p.wcut        = 0;                // Off-shell mass (width) cut
191:
192:      // Neutral mesons/baryons get 0 for their name
193:      if (std::abs(p.chargeX3) == 0 && p.pdg > 100) {
194:        p.name = p.name + "0";
195:      } else if ((p.chargeX3 != 0) &&
196:                 !((p.pdg >= 1 && p.pdg <= 6) ||
197:                   (p.pdg == 12 || p.pdg == 14 || p.pdg == 16))) {  // quarks & neutrinos
198:
199:        std::string signstr = p.chargeX3 > 0 ? std::string(std::abs(p.chargeX3 / 3), '+')
200:                                             : std::string(std::abs(p.chargeX3 / 3), '-');
201:
202:        p.name = name + signstr;
203:      }
204:
205:      // -------------------------------------------------
206:      // SPECIAL CASES (not following spin numbering)
207:
208:      // SM bosons
209:      if (p.pdg == 21) {  // gluon
210:        p.spinX2 = 2;
211:        p.P      = -1;
212:        p.C      = 0;  // Not defined
213:      }
214:      if (p.pdg == 22) {  // gamma
215:        p.spinX2 = 2;
216:        p.P      = -1;
217:        p.C      = -1;
218:      }
219:      if (std::abs(p.pdg) == 24) {  // W+-
220:        p.spinX2 = 2;
221:        p.P      = 0;  // Not defined
222:        p.C      = 0;  // Not defined
223:      }
224:      if (p.pdg == 23) {  // Z
225:        p.spinX2 = 2;
226:        p.P      = 0;  // Not defined
227:        p.C      = 0;  // Not defined
228:      }
229:      if (p.pdg == 25) {  // H
230:        p.spinX2 = 0;
231:        p.P      = 1;  // Scalar SM higgs
232:        p.C      = 1;  // Scalar SM higgs
233:      }
234:
235:      // SM fermions
236:      if (p.pdg >= 11 && p.pdg <= 16) {  // leptons and neutrinos
237:        p.spinX2 = 1;
238:        p.P      = 1;
239:        p.C      = 0;  // Not defined
240:      }
241:      if (p.pdg >= 1 && p.pdg <= 6) {  // quarks
242:        p.spinX2 = 1;
243:        p.P      = 1;
244:        p.C      = 0;  // Not defined
245:      }
246:
247:      // -------------------------------------------------
248:      // Meson J^PC (L) assignments
249:
```



```
250:      if (p.spinX2 == 0 || p.spinX2 == 2 || p.spinX2 == 4 || p.spinX2 == 8 ||
251:          p.spinX2 == 10) {  // Is a boson
252:
253:        if (std::to_string(p.pdg).length() == 3 || std::to_string(p.pdg).length() == 5 ||
254:            std::to_string(p.pdg).length() == 6 || std::to_string(p.pdg).length() == 7) {
255:          unsigned int N = std::to_string(p.pdg).length();
256:          unsigned int m = 0;
257:
258:          if (N == 5) { m = 0; }
259:          if (N == 6) { m = 1; }
260:          if (N == 7) { m = 2; }
261:
262:          // [code   JPC   L]
263:
264:          // L = J-1, S = 1
265:          // -----------------
266:
267:          if (N == 3 || (std::to_string(p.pdg)[m] == '0' && std::to_string(p.pdg)[m + 1] == '0')) {
268:            // 00qq3   1--   0
269:            if (std::to_string(p.pdg)[N - 1] == '3') p.setPCL(-1, -1, 0);
270:
271:            // 00qq5   2++   1
272:            if (std::to_string(p.pdg)[N - 1] == '5') p.setPCL(1, 1, 1);
273:
274:            // 00qq7   3--   2
275:            if (std::to_string(p.pdg)[N - 1] == '7') p.setPCL(-1, -1, 2);
276:
277:            // 00qq9   4++   3
278:            if (std::to_string(p.pdg)[N - 1] == '9') p.setPCL(1, 1, 3);
279:          }
280:
281:          // L = J, S = 0
282:          // -----------------
283:
284:          // 00qq1   0-+   0
285:          if (N == 3 || (std::to_string(p.pdg)[m] == '0' && std::to_string(p.pdg)[m + 1] == '0')) {
286:            if (std::to_string(p.pdg)[N - 1] == '1') p.setPCL(-1, 1, 0);
287:          }
288:
289:          if (N >= 5) {
290:            // 10qq3   1+-   1
291:            if (std::to_string(p.pdg)[m] == '1' && std::to_string(p.pdg)[N - 1] == '3')
292:              p.setPCL(1, -1, 1);
293:
294:            // 10qq5   2-+   2
295:            if (std::to_string(p.pdg)[m] == '1' && std::to_string(p.pdg)[N - 1] == '5')
296:              p.setPCL(-1, 1, 2);
297:
298:            // 10qq7   3+-   3
299:            if (std::to_string(p.pdg)[m] == '1' && std::to_string(p.pdg)[N - 1] == '7')
300:              p.setPCL(+1, -1, 3);
301:
302:            // 10qq9   4-+   4
303:            if (std::to_string(p.pdg)[m] == '1' && std::to_string(p.pdg)[N - 1] == '9')
304:              p.setPCL(-1, 1, 4);
305:
306:            // L = J, S = 1
307:            // -----------------
308:
309:            // 20qq3   1++   1
310:            if (std::to_string(p.pdg)[m] == '2' && std::to_string(p.pdg)[N - 1] == '3')
311:              p.setPCL(1, 1, 1);
312:
313:            // 20qq5   2--   2
314:            if (std::to_string(p.pdg)[m] == '2' && std::to_string(p.pdg)[N - 1] == '5')
315:              p.setPCL(-1, -1, 2);
316:
317:            // 20qq7   3++   3
318:            if (std::to_string(p.pdg)[m] == '2' && std::to_string(p.pdg)[N - 1] == '7')
319:              p.setPCL(1, 1, 3);
320:
321:            // 20qq9   4--   4
322:            if (std::to_string(p.pdg)[m] == '2' && std::to_string(p.pdg)[N - 1] == '9')
323:              p.setPCL(-1, -1, 4);
324:
325:            // L = J+1, S = 1
326:            // -----------------
327:
328:            // 10qq1   0++   1
329:            if (std::to_string(p.pdg)[m] == '1' && std::to_string(p.pdg)[N - 1] == '1')
330:              p.setPCL(1, 1, 1);
331:
332:            // 30qq3   1--   2
```



```
333:            if (std::to_string(p.pdg)[m] == '3' && std::to_string(p.pdg)[N - 1] == '3')
334:              p.setPCL(-1, -1, 2);
335:
336:            // 30qq5   2++   3
337:            if (std::to_string(p.pdg)[m] == '3' && std::to_string(p.pdg)[N - 1] == '5')
338:              p.setPCL(1, 1, 3);
339:
340:            // 30qq7   3--   4
341:            if (std::to_string(p.pdg)[m] == '3' && std::to_string(p.pdg)[N - 1] == '7')
342:              p.setPCL(-1, -1, 4);
343:
344:            // 30qq9   4++   5
345:            if (std::to_string(p.pdg)[m] == '3' && std::to_string(p.pdg)[N - 1] == '9')
346:              p.setPCL(1, 1, 5);
347:          }
348:        }
349:      }
350:
351:      // -------------------------------------------------
352:      // Baryon J^PC (L) assignments
353:
354:      if (p.spinX2 == 1 || p.spinX2 == 3 || p.spinX2 == 5 || p.spinX2 == 7 ||
355:          p.spinX2 == 9) {  // Is a fermion
356:
357:        if (std::to_string(p.pdg).length() == 4 || std::to_string(p.pdg).length() == 5 ||
358:            std::to_string(p.pdg).length() == 6 || std::to_string(p.pdg).length() == 7) {
359:          // [code   J^P]
360:          unsigned int N = std::to_string(p.pdg).length();
361:          unsigned int m = 0;
362:
363:          if (N == 6) { m = 0; }
364:          if (N == 7) { m = 1; }
365:
366:          // 00qqq2
367:          if (N == 4 || (std::to_string(p.pdg)[m] == '0' && std::to_string(p.pdg)[m + 1] == '0')) {
368:            if (std::to_string(p.pdg)[N - 1] == '2') p.setPCL(1, 0, 0);
369:          }
370:          // 20qqq2
371:          if (std::to_string(p.pdg)[m] == '2' && std::to_string(p.pdg)[m + 1] == '0') {
372:            if (std::to_string(p.pdg)[N - 1] == '2') p.setPCL(1, 0, 0);
373:          }
374:          // 21qqq2
375:          if (std::to_string(p.pdg)[m] == '2' && std::to_string(p.pdg)[m + 1] == '1') {
376:            if (std::to_string(p.pdg)[N - 1] == '2') p.setPCL(1, 0, 0);
377:          }
378:          // 10qqq2
379:          if (std::to_string(p.pdg)[m] == '1' && std::to_string(p.pdg)[m + 1] == '0') {
380:            if (std::to_string(p.pdg)[N - 1] == '2') p.setPCL(-1, 0, 1);
381:          }
382:
383:          // 00qq4
384:          if (N == 4 || (std::to_string(p.pdg)[m] == '0' && std::to_string(p.pdg)[m + 1] == '0')) {
385:            if (std::to_string(p.pdg)[N - 1] == '4') p.setPCL(1, 0, 0);
386:          }
387:          // 20qqq4
388:          if (std::to_string(p.pdg)[m] == '2' && std::to_string(p.pdg)[m + 1] == '0') {
389:            if (std::to_string(p.pdg)[N - 1] == '4') p.setPCL(1, 0, 0);
390:          }
391:          // 11qqq2
392:          if (std::to_string(p.pdg)[m] == '1' && std::to_string(p.pdg)[m + 1] == '1') {
393:            if (std::to_string(p.pdg)[N - 1] == '2') p.setPCL(1, 0, 1);
394:          }
395:          // 12qqq4
396:          if (std::to_string(p.pdg)[m] == '1' && std::to_string(p.pdg)[m + 1] == '2') {
397:            if (std::to_string(p.pdg)[N - 1] == '4') p.setPCL(-1, 0, 1);
398:          }
399:
400:          // Check this case, PDG seems ambiguous here
401:          if (N == 5) { p.setPCL(1, 0, 0); }
402:        }
403:      }
404:
405:      // -------------------------------------------------
406:      // Finally, ADD TO THE TABLE
407:      PDG_table.insert(std::make_pair(p.pdg, p));
408:
409:      // ANTIPARTICLE (id.size < 3 because 0,+,++)
410:      if (((int)std::abs(p.spinX2 / 2.0) != (double)std::abs(p.spinX2 / 2.0) && id.size() < 3) ||
411:          (std::abs(p.chargeX3) > 0 && id.size() < 3)) {
412:        gra::MParticle antip = p;
413:
414:        // antiquarks & antineutrinos get tilde
415:        if ((1 <= p.pdg && p.pdg <= 6) || (p.pdg == 12 || p.pdg == 14 || p.pdg == 16)) {
```

```
416:           antip.name = name + "~";
417:         } else if (p.chargeX3 != 0) {
418:           std::string signstr = p.chargeX3 > 0 ? std::string(std::abs(p.chargeX3 / 3), '-')
419:                                                : std::string(std::abs(p.chargeX3 / 3), '+');
420:
421:           antip.name = name + signstr;
422:         } else if (p.chargeX3 == 0) {  // e.g. anti K0(S)
423:           antip.name = name + "0";
424:         }
425:
426:         antip.pdg      = -p.pdg;
427:         antip.chargeX3 = -p.chargeX3;
428:
429:         // Flip the parity for fermions
430:         if (p.spinX2 == 1 || p.spinX2 == 3 || p.spinX2 == 5 || p.spinX2 == 7 || p.spinX2 == 9) {
431:           antip.P = -p.P;
432:         }
433:         // Do not flip for the bosons
434:
435:         // ADD TO THE TABLE
436:         PDG_table.insert(std::make_pair(antip.pdg, antip));
437:       }
438:     }
439:   } // PDG file line while loop
440:
441:   // Process now here fast variable setup >>
442:   std::cout << rang::fg::green << "[DONE]" << rang::fg::reset << std::endl;
443: }
444:
445: // Recursive function to read the decay process string
446: void MPDG::TokeniseProcess(const std::string &str, int depth,
447:                            std::vector<gra::MDecayBranch> &branches) const {
448:   std::string mother;
449:   int         aM = 0;
450:
451:   for (std::string::size_type i = 0; i < str.size(); ++i) {
452:     // printf("%d,%d| \n", depth, i);
453:
454:     // If no subdecays left, extract particles directly
455:     std::string substr = str.substr(i, str.size() - 1);
456:     if (!IsDecay(substr)) {
457:       std::vector<std::string> particles = gra::aux::Extract(substr);
458:       for (std::size_t k = 0; k < particles.size(); ++k) {
459:         // std::cout << "L" << depth << " " <<
460:         // particles[k] << std::endl;
461:
462:         // **************
463:         // Create decay branch
464:         gra::MDecayBranch branch;
465:
466:         branch.p      = FindByPDGName(particles[k]);
467:         branch.depth = depth;
468:         branches.push_back(branch);
469:         // **************
470:       }
471:       return;
472:     }
473:
474:     // Found sub decay '>', then find closing brackets
475:     if (str[i] == '>') {
476:       mother = str.substr(aM, i - aM);
477:
478:       // Check if mother is actually several particles,
479:       // only the last is mother, others before the last are
480:       // not
481:       // (say e+ e- rho0 > (pi+ pi-) ), mother is rho0
482:       std::vector<std::string> particles = gra::aux::Extract(mother);
483:       for (std::size_t k = 0; k < particles.size(); ++k) {
484:         std::cout << "L" << depth << " " << particles[k] << std::endl;
485:
486:         // **************
487:         // Create decay branch
488:         gra::MDecayBranch branch;
489:
490:         branch.p      = FindByPDGName(particles[k]);
491:         branch.depth = depth;
492:         branches.push_back(branch);
493:         // **************
494:       }
495:
496:       mother = particles[particles.size() - 1];  // Choose the last one
497:
498:       // Find brackets (  )
```



```
499:       std::vector<std::string::size_type> L;
500:       std::vector<std::string::size_type> R;
501:
502:       // Now find the
503:       unsigned int a = 0;
504:       unsigned int b = 0;
505:       for (std::string::size_type j = i + 1; j < str.size(); ++j) {
506:         if (str[j] == '(') { L.push_back(j); }
507:         if (str[j] == ')') { R.push_back(j); }
508:         if (L.size() == R.size()) && L.size() > 0) {  // Found outer closing brackets
509:           a = L[0];
510:           b = R[R.size() - 1];
511:
512:           // Take the token
513:           std::string token = str.substr(a + 1, b - a - 1);  // pos, len
514:           // printf("a = %d, b = %d \n", a,b );
515:           TokeniseProcess(token, depth + 1, branches[branches.size() - 1].legs);  // Recursion
516:           i  = b + 1;
517:           aM = b + 1;
518:           break;
519:         }
520:       }
521:     }
522:   }
523: }
524:
525: // Check if string contains '>'
526: bool MPDG::IsDecay(const std::string &str) const {
527:   bool found = false;
528:   for (std::string::size_type i = 0; i < str.size(); ++i) {
529:     if (str[i] == '>') { found = true; }
530:   }
531:   return found;
532: }
533:
534: // Print out PDG table
535: void MPDG::PrintPDGTable() const {
536:   std::cout << "MPDG::PrintPDGTable:" << std::endl << std::endl;
537:   printf("\t\tpdg\tmass\t\twidth\t\tcharge\tJ^PC\tname\n");
538:
539:   std::map<int, gra::MParticle>::const_iterator it = PDG_table.begin();
540:
541:   unsigned int counter = 0;
542:   while (it != PDG_table.end()) {
543:     const gra::MParticle p = it->second;
544:
545:     printf("%d\t\t0d\t\0.6f\t\0.6f\t\2s\t\s\s\s\t\s \n", ++counter, p.pdg, p.mass, p.width,
546:            gra::aux::Charge3XtoString(p.chargeX3).c_str(),
547:            gra::aux::Spin2XtoString(p.spinX2).c_str(), gra::aux::ParityToString(p.P).c_str(),
548:            gra::aux::FarityToString(p.C).c_str(), p.name.c_str());
549:     ++it;
550:   }
551: }
552:
553: // Find particle by PDG ID
554: const gra::MParticle &MPDG::FindByPDG(int pdgcode) const {
555:   std::map<int, gra::MParticle>::const_iterator it = PDG_table.find(pdgcode);
556:   if (it != PDG_table.end()) { return it->second; }
557:
558:   // Throw a fatal error, did not found the particle
559:   PrintPDGTable();
560:   std::string str = "MProcess::FindByPDG: Unknown PDG ID: " + std::to_string(pdgcode);
561:   throw std::invalid_argument(str);
562: }
563:
564: // Find particle by PDG name
565: const gra::MParticle &MPDG::FindByPDGName(const std::string &pdgname) const {
566:   std::map<int, gra::MParticle>::const_iterator it = PDG_table.begin();
567:   while (it != PDG_table.end()) {
568:     if (it->second.name.compare(pdgname) == 0) { return it->second; }
569:     ++it;
570:   }
571:
572:   // Check if we have PDG number input (we allow that too)
573:   if (gra::aux::IsIntegerDigits(pdgname)) {
574:     const int pdgcode = std::stoi(pdgname);
575:     return FindByPDG(pdgcode);
576:   }
577:
578:   // Throw a fatal error, did not found the particle
579:   PrintPDGTable();
580:   std::string str = "MProcess::FindByPDGName: Unknown PDG name: " + pdgname;
581:   throw std::invalid_argument(str);
```



```
582: }
583:
584: }  // namespace gra
585:
```



```
  1: // Random number class
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: // C++
  7: #include <complex>
  8: #include <random>
  9: #include <vector>
 10:
 11: // Own
 12: #include "Graniitti/MMath.h"
 13: #include "Graniitti/MRandom.h"
 14:
 15: namespace gra {
 16: // Poisson distributed random numbers with mean lambda
 17: int MRandom::PoissonRandom(double lambda) {
 18:   const unsigned int ALGO = 2;
 19:
 20:   // Knuth algorithm
 21:   if (ALGO == 1) {
 22:     const double L = std::exp(-lambda);
 23:     int          k = 0;
 24:     double       p = 1;
 25:
 26:     do {
 27:       ++k;
 28:       p *= U(0, 1);
 29:     } while (p > L);
 30:     return k - 1;
 31:   }
 32:   // Inversion search algorithm
 33:   if (ALGO == 2) {
 34:     int    k = 0;
 35:     double p = std::exp(-lambda);
 36:     double s = p;
 37:     double u = U(0, 1);
 38:
 39:     while (u > s) {
 40:       ++k;
 41:       p = p * lambda / k;
 42:       s += p;
 43:     }
 44:     return k;
 45:   }
 46: }
 47:
 48: // Exponential random numbers with rate lambda (mean is given by 1/lambda)
 49: double MRandom::ExpRandom(double lambda) {
 50:   const double R = U(0, 1);
 51:   return -1.0 / lambda * std::log(1.0 - R);
 52: }
 53:
 54: // Powerlaw random numbers from [a,b] with exponent alpha (e.g. -2 for ~ 1/x^2)
 55: double MRandom::PowerRandom(double a, double b, double alpha) {
 56:   const double min_pow = std::pow(a, alpha + 1.0);
 57:   const double max_pow = std::pow(b, alpha + 1.0);
 58:   return std::pow((max_pow - min_pow) * U(0, 1) + min_pow, 1.0 / (alpha + 1));
 59: }
 60:
 61: // Uniform random numbers from [a,b)
 62: double MRandom::U(double a, double b) {
 63:   const double value = flat(rng);
 64:   return a + (b - a) * value;
 65: }
 66:
 67: // Gaussian random numbers from normal distribution with mean mu and std sigma
 68: double MRandom::G(double mu, double sigma) {
 69:   const double value = gaussian(rng);
 70:   return mu + sigma * value;
 71: }
 72:
 73: // Relativistic Breit-Wigner sampling f(s) ~ 1/[(s-m^2)^2 + m^2Gamma^2]
 74: //
 75: // Input: m0     = Pole mass (GeV)
 76: //        Gamma  = Full width (GeV)
 77: //        LIMIT  = gives m0 +- LIMIT * GAMMA,
 78: //        M_MIN  = optional minimum bound
 79: //
 80: // Return mass (GeV)
 81: //
 82: double MRandom::RelativisticBWRandom(double m0, double Gamma, double LIMIT, double M_MIN) {
 83:   // No width case
```

```
  84:   if (Gamma < 1e-40) { return m0; }
  85:
  86:   const double m2    = math::pow2(m0);
  87:   const double m2max = gra::math::pow2(m0 + LIMIT * Gamma);
  88:   const double m2min =
  89:       std::max(std::max(0.0, math::pow2(M_MIN)), gra::math::pow2(m0 - LIMIT * Gamma));
  90:
  91:   double m2val = 0.0;
  92:   while (true) {
  93:     const double R = U(0, 1);
  94:     m2val =
  95:         m2 +
  96:         m0 * Gamma *
  97:             std::tan(std::atan2(m2min - m2, m0 * Gamma) +
  98:                      R * (std::atan2(m2max - m2, m0 * Gamma) - std::atan2(m2min - m2, m0 * Gamma)));
  99:
 100:     // Note >= handles massless case, otherwise stuck with m = 0
 101:     if (m2val >= 0) { break; }
 102:   }
 103:   return gra::math::msqrt(m2val);
 104: }
 105:
 106: // Cauchy (non-relativistic Breit-Wigner) sampling
 107: //
 108: // Input as with RelativisticBWRandom
 109: //
 110: double MRandom::CauchyRandom(double m0, double Gamma, double LIMIT, double M_MIN) {
 111:   if (Gamma < 1e-40) { return m0; }
 112:
 113:   const double mmax = m0 + LIMIT * Gamma;
 114:   const double mmin = std::max(std::max(0.0, M_MIN), m0 - LIMIT * Gamma);
 115:
 116:   double mval = 0.0;
 117:   while (true) {
 118:     const double R     = U(0, 1);
 119:     const double value = (Gamma * 0.5) * std::tan(gra::math::PI * (R - 0.5)) + m0;
 120:
 121:     if (value < mmax && value > mmin) {
 122:       mval = value;
 123:       break;
 124:     }
 125:   }
 126:   return mval;
 127: }
 128:
 129: // K-dimensional Dirichlet distribution with parameter vector alpha of length K
 130: void MRandom::DirRandom(const std::vector<double> &alpha, std::vector<double> &y) {
 131:   unsigned int K = alpha.size();
 132:
 133:   // Draw K independent samples from Gamma(shape = alpha_i, scale = 1)
 134:   y.resize(K, 0.0);
 135:   for (std::size_t i = 0; i < K; ++i) {
 136:     std::gamma_distribution<double> gammarnd(alpha[i], 1.0);
 137:     y[i] = gammarnd(rng);  // Draw sample
 138:   }
 139:   // Take the sum
 140:   double y_sum = 0.0;
 141:   for (std::size_t i = 0; i < K; ++i) { y_sum += y[i]; }
 142:   // Normalize
 143:   for (std::size_t i = 0; i < K; ++i) { y[i] /= y_sum; }
 144: }
 145:
 146: // Negative binomial distribution with parameters avgN and k
 147: double MRandom::NBDpdf(int n, double avgN, double k) {
 148:   // return Cbinom(n+k-1,n)
 149:   return tgamma(n + k) / (tgamma(n + 1) * tgamma(k)) * std::pow(avgN / (k + avgN), n) *
 150:          std::pow(k / (k + avgN), k);
 151: }
 152:
 153: // Random sample from NBD distribution with parameters avgN and k
 154: int MRandom::NBDRandom(double avgN, double k, int maxvalue) {
 155:   const unsigned int MAXTRIAL = 1e7;
 156:
 157:   // Random integer from [0,NBins-1]
 158:   std::uniform_int_distribution<int> RANDI(1, maxvalue);
 159:
 160:   // Acceptance-Rejection
 161:   unsigned int trials = 0;
 162:   while (true) {
 163:     const int    n   = RANDI(rng);
 164:     const double val = NBDpdf(n, avgN, k);
 165:     if (U(0, 1) < val) { return n; }
 166:     ++trials;
```



```
 167:     if (trials > MAXTRIAL) {
 168:       return avgN;  // Return the mean value
 169:     }
 170:   }
 171: }
 172:
 173: // Log-distribution with with parameter p
 174: double MRandom::Logpdf(int k, double p) {
 175:   return -1.0 / std::log(1 - p) * std::pow(p, k) / static_cast<double>(k);
 176: }
 177:
 178: // Random sample from log-distribution with parameter p
 179: int MRandom::LogRandom(double p, int maxvalue) {
 180:   // Random integer from [0,NBins-1]
 181:   std::uniform_int_distribution<int> RANDI(0, maxvalue - 1);
 182:
 183:   // Acceptance-Rejection
 184:   while (true) {
 185:     const int    n   = RANDI(rng);
 186:     const double val = Logpdf(n, p);
 187:     if (U(0, 1) < val) { return n; }
 188:   }
 189: }
 190:
 191: }  // namespace gra
 192:
```



```
   1: // Tensor Pomeron amplitudes
   2: //
   3: //
   4: // [REFERENCE: Ewerz, Maniatis, Nachtmann, arxiv.org/abs/1309.3478]
   5: // [REFERENCE: Lebiodowicz, Nachtmann, Szczurek, arxiv.org/abs/1601.04537]
   6: // [REFERENCE: LNS, arxiv.org/abs/1801.03902]
   7: // [REFERENCE: LNS, arxiv.org/abs/1901.11490]
   8: //
   9: // (c) 2017-2020 Mikael Mieskolainen
  10: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  11:
  12: // C++
  13: #include <complex>
  14: #include <iostream>
  15: #include <vector>
  16:
  17: // Own
  18: #include "Granitti/M4Vec.h"
  19: #include "Granitti/MAux.h"
  20: #include "Granitti/MDirac.h"
  21: #include "Granittti/MForm.h"
  22: #include "Granitti/MKinematics.h"
  23: #include "Granitti/MMath.h"
  24: #include "Granitti/MQED.h"
  25: #include "Granitti/MRegge.h"
  26: #include "Granitti/MTensorPomeron.h"
  27:
  28: // FTensor algebra
  29: #include "FTensor.hpp"
  30:
  31: // LOOP MACROS
  32: #define FOR_EACH_2(X)       \
  33:   for (const auto &u : X) { \
  34:     for (const auto &v : X) {
  35: #define FOR_EACH_2_END \
  36:   }                    \
  37:   }
  38:
  39: #define FOR_EACH_3(X)         \
  40:   for (const auto &u : X) {   \
  41:     for (const auto &v : X) { \
  42:       for (const auto &k : X) {
  43: #define FOR_EACH_3_END \
  44:   }                    \
  45:   }                    \
  46:   }
  47:
  48: #define FOR_EACH_4(X)           \
  49:   for (const auto &u : X) {     \
  50:     for (const auto &v : X) {   \
  51:       for (const auto &k : X) { \
  52:         for (const auto &l : X) {
  53: #define FOR_EACH_4_END \
  54:   }                    \
  55:   }                    \
  56:   }                    \
  57:   }
  58:
  59: #define FOR_EACH_5(X)             \
  60:   for (const auto &u : X) {       \
  61:     for (const auto &v : X) {     \
  62:       for (const auto &k : X) {   \
  63:         for (const auto &l : X) { \
  64:           for (const auto &r : X) {
  65: #define FOR_EACH_5_END \
  66:   }                    \
  67:   }                    \
  68:   }                    \
  69:   }                    \
  70:   }
  71:
  72: #define FOR_EACH_6(X)               \
  73:   for (const auto &u : X) {         \
  74:     for (const auto &v : X) {       \
  75:       for (const auto &k : X) {     \
  76:         for (const auto &l : X) {   \
  77:           for (const auto &r : X) { \
  78:             for (const auto &s : X) {
  79: #define FOR_EACH_6_END \
  80:   }                    \
  81:   }                    \
  82:   }                    \
  83:   }
```



```
  84:   }                    \
  85:   }
  86:
  87: #define FOR_PP_HELICITY             \
  88:   for (const auto &ha : {0, 1}) {   \
  89:     for (const auto &hb : {0, 1}) { \
  90:       for (const auto &h1 : {0, 1}) { \
  91:         for (const auto &h2 : {0, 1}) {
  92: #define FOR_PP_HELICITY_END \
  93:   }                         \
  94:   }                         \
  95:   }                         \
  96:   }
  97:
  98: using gra::aux::indices;
  99: using gra::math::PI;
 100: using gra::math::msqrt;
 101: using gra::math::pow2;
 102: using gra::math::zi;
 103:
 104: using FTensor::Tensor0;
 105: using FTensor::Tensor1;
 106: using FTensor::Tensor2;
 107: using FTensor::Tensor3;
 108: using FTensor::Tensor4;
 109:
 110: namespace gra {
 111:
 112: // Constructor
 113: MTensorPomeron::MTensorPomeron(gra::LORENTZSCALAR &lts, const std::string &modelfile) {
 114:   CalcRTensor();  // Pre-Calculate tensors
 115:
 116:   // @@ MULTITHREADING LOCK NEEDED FOR THE INITIALIZATION @@
 117:   gra::g_mutex.lock();
 118:
 119:   if (!Param.initialized) {
 120:     try {
 121:       Param.ReadParameters(modelfile);
 122:       PARAM_REGGE::ReadParameters(0, modelfile);
 123:
 124:     } catch (...) {
 125:       gra::g_mutex.unlock();  // need to release here, otherwise get infinite lock
 126:       throw;
 127:     }
 128:   }
 129:   gra::g_mutex.unlock();
 130: }
 131:
 132:
 133: // Return decay coupling constant for resonance (M,Gamma) with decay daughter mass mf
 134: // BR being the branching ratio BR = Width_partial / Width_total
 135: //
 136: // Decay: Mother (spin = 0/1/2) -> scalar or pseudoscalar daughters
 137: //
 138: double MTensorPomeron::GDecay(int J, double M, double Gamma, double mf, double BR) {
 139:   const double S0          = 1.0;  // Should be set as the same scale as in decay amplitudes iG[]
 140:   const double partialWidth = Gamma * BR;
 141:
 142:   const double P = sqrt(1 - 4 * pow2(mf / M));
 143:
 144:   if (J == 0) {
 145:     return sqrt(partialWidth / (1.0 / (16 * PI * M) * pow2(S0) * P));
 146:   } else if (J == 1) {
 147:     return sqrt(partialWidth / (M / (192 * PI) * math::pow3(P)));
 148:   } else if (J == 2) {
 149:     return sqrt(partialWidth / (M / (480 * PI) * pow2(M / S0) * math::pow5(P)));
 150:   } else if (J == 4) {
 151:     return 1.0;  // future
 152:   } else {
 153:     throw std::invalid_argument("MTensorPomeron::GDecay: Unknown input spin J = " +
 154:                                 std::to_string(J));
 155:   }
 156: }
 157:
 158:
 159: // 2 -> 3 amplitudes
 160: //
 161: // return value: matrix element squared with helicities summed and averaged over
 162: // lts.hamp:     individual complex helicity amplitudes
 163: //
 164: double MTensorPomeron::ME3(gra::LORENTZSCALAR &lts) const {
 165:   // Free Lorentz indices [second parameter denotes the range of index]
 166:   FTensor::Index<'a', 4> mu1;
```

```
167:   FTensor::Index<'b', 4> nu1;
168:   FTensor::Index<'c', 4> rho1;
169:   FTensor::Index<'d', 4> rho2;
170:   FTensor::Index<'g', 4> alpha1;
171:   FTensor::Index<'h', 4> beta1;
172:   FTensor::Index<'i', 4> alpha2;
173:   FTensor::Index<'j', 4> beta2;
174:   FTensor::Index<'k', 4> mu2;
175:   FTensor::Index<'l', 4> nu2;
176:
177:   // Kinematics
178:   const M4Vec pa = lts.pbeam1;
179:   const M4Vec pb = lts.pbeam2;
180:
181:   const M4Vec p1 = lts.pfinal[1];
182:   const M4Vec p2 = lts.pfinal[2];
183:   const M4Vec p3 = lts.decaytree[0].p4;
184:   const M4Vec p4 = lts.decaytree[1].p4;
185:
186:   // ------------------------------------------------------------------
187:
188:   // Spinors (2 helicities)
189:   const std::array<std::vector<std::complex<double>>, 2> u_a = SpinorStates(pa, "u");
190:   const std::array<std::vector<std::complex<double>>, 2> u_b = SpinorStates(pb, "u");
191:
192:   const std::array<std::vector<std::complex<double>>, 2> ubar_1 = SpinorStates(p1, "ubar");
193:   const std::array<std::vector<std::complex<double>>, 2> ubar_2 = SpinorStates(p2, "ubar");
194:
195:   // ------------------------------------------------------------------
196:
197:   // t-channel Pomeron propagators
198:   const Tensor4<std::complex<double>, 4, 4, 4, 4> iDP_1 = iD_P(lts.s1, lts.t1);
199:   const Tensor4<std::complex<double>, 4, 4, 4, 4> iDP_2 = iD_P(lts.s2, lts.t2);
200:
201:   // High Energy Limit proton-Pomeron-proton spinor structure
202:   const Tensor2<std::complex<double>, 4, 4> iG_1a = iG_PppHE(p1, pa);
203:   const Tensor2<std::complex<double>, 4, 4> iG_2b = iG_PppHE(p2, pb);
204:
205:   // ------------------------------------------------------------------
206:
207:   // Reset helicity amplitude container
208:   // Init with zeros, important for the coherent sum!
209:   lts.hamp                = std::vector<std::complex<double>>(64, 0.0);
210:   std::size_t maxindex = 0;
211:
212:   // >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
213:   // 2. Coherent sum of Resonances (loop over)
214:   for (auto &x : lts.RESONANCES) {
215:     const PARAM_RES res = x.second;
216:
217:     // Resonance parameters
218:     const double M0    = res.p.mass;
219:     const double Gamma = res.p.width;
220:     const int    J     = res.p.spinX2 / 2.0;
221:     const int    P     = res.p.P;
222:
223:     // ------------------------------------------------------------------
224:
225:     // ==================================================================
226:     // Pomeron-Pomeron-Scalar structure
227:     //
228:     //
229:     if (J == 0 && P == 1) {
230:       Tensor4<std::complex<double>, 4, 4, 4, 4> cvtx;
231:       std::vector<std::complex<double>>          eps3eps4;
232:
233:       cvtx = iG_PPS_total(lts.q1, lts.q2, M0, "scalar", res.g_Tensor);
234:
235:       // Scalar BW-propagator
236:       const std::complex<double> iD = iD_MES(lts.pfinal[0], M0, Gamma);
237:
238:       // [PS PS]  Pseudoscalar pair decay
239:       if (lts.decaytree[0].p.spinX2 == 0 && lts.decaytree[1].p.spinX2 == 0) {
240:         eps3eps4.clear();
241:         eps3eps4.push_back(
242:             iG_f0ss(lts.decaytree[0].p4, lts.decaytree[1].p4, M0, res.hel.g_decay_tensor[0]));
243:       }
244:
245:       // [V   V]  Massive vector pair decay
246:       else if (lts.decaytree[0].p.spinX2 == 2 && lts.decaytree[1].p.spinX2 == 2) {
247:         if (res.hel.g_decay_tensor.size() != 2) {
248:           throw std::invalid_argument(
249:               "MTensorPomeron:: S->VV Coupling array [size 2] 'hel.g_decay_tensor' not in "
```



```
250:               "BRANCHING.json for resonance PDG = " +
251:               std::to_string(res.p.pdg));
252:         }
253:         // Decay vertex with different outgoing helicity combinations
254:         Tensor2<std::complex<double>, 4, 4> iGf0vv =
255:             iG_f0vv(p3, p4, M0, res.hel.g_decay_tensor[0], res.hel.g_decay_tensor[1]);
256:         eps3eps4 = MassiveSpin1Pol1Sum(iGf0vv, p3, p4);
257:
258:         // Sequential decay correlations [TBD]
259:         // const Tensor2<std::complex<double>, 4, 4> iGvv2psps = iG_vv2psps({},
260:         // lts.decaytree[0].p.pdg);  eps3eps4.clear();  eps3eps4.push_back( iGf0vv(mu1, mu2) *
261:         // iGvv2psps(mu1, mu2) );
262:
263:       } else {
264:         throw std::invalid_argument("MTensorPomeron::ME3: Unknown decay mode for scalar resonance");
265:       }
266:
267:       // Two helicity states for incoming and outgoing protons
268:       std::size_t index = 0;
269:       FOR_PP_HELICITY;
270:
271:       // Apply proton leg helicity conservation / No helicity flip (high energy limit)
272:       if (ha != h1 || hb != h2) { continue; }
273:
274:       // Full proton-Pomeron-proton spinor structure (upper and lower vertex)
275:       // const Tensor2<std::complex<double>,4,4> iG_1a = iG_Ppp(p1, pa, ubar_1[h1], u_a[ha]);
276:       // const Tensor2<std::complex<double>,4,4> iG_2b = iG_Ppp(p2, pb, ubar_2[h2], u_b[hb]);
277:
278:       // Evaluate block
279:       const std::complex<double> block = (-zi) * iG_1a(mu1, nu1) * iDP_1(mu1, nu1, alpha1, beta1) *
280:                                          cvtx(alpha1, beta1, alpha2, beta2) * iD *
281:                                          iDP_2(alpha2, beta2, mu2, nu2) * iG_2b(mu2, nu2);
282:
283:       // s-channel amplitude for each outgoing helicity combination
284:       for (const auto &ind : indices(eps3eps4)) {
285:         const std::complex<double> A = block * eps3eps4[ind];
286:         lts.hamp[index] += A;  // Note +=
287:         ++index;
288:       }
289:
290:       FOR_PP_HELICITY_END;
291:       maxindex = index > maxindex ? index : maxindex;
292:
293:       // ==================================================================
294:       // Pomeron-Gamma-Vector (Photoproduction of rho, phi ...) structure
295:       //
296:       // Should add vector meson mass dependent running of t-form factors
297:       //
298:       // p --x---------------- P
299:       //      *
300:       //      y *   rho0
301:       //         *x=====x===== rho0
302:       //                 |
303:       //                 |  P
304:       //                 |
305:       // p ------------x------ P
306:       //
307:     } else if (J == 1 && P == -1) {
308:       // Gamma-Vector coupling
309:       const Tensor2<std::complex<double>, 4, 4> iGyV = iG_yV(0, res.p.pdg);
310:
311:       // Gamma propagators
312:       const Tensor2<std::complex<double>, 4, 4> iDy_1 = iD_y(lts.q1.M2());
313:       const Tensor2<std::complex<double>, 4, 4> iDy_2 = iD_y(lts.q2.M2());
314:
315:       // Vector-meson propagators
316:       const bool                                INDEX_UP = true;
317:       const Tensor4<std::complex<double>, 4, 4> iDV_1    = iD_VMES(lts.q1, M0, Gamma, INDEX_UP);
318:       const Tensor4<std::complex<double>, 4, 4> iDV   = iD_VMES(lts.pfinal[0], M0, Gamma, INDEX_UP);
319:       const Tensor4<std::complex<double>, 4, 4> iDV_2 = iD_VMES(lts.q2, M0, Gamma, INDEX_UP);
320:
321:       // Pomeron-Vector-Vector coupling
322:       const Tensor4<std::complex<double>, 4, 4, 4, 4> iGPvv_1 =
323:           iG_Pvv(lts.pfinal[0], lts.q1, res.g_Tensor[0], res.g_Tensor[1]);
324:       const Tensor4<std::complex<double>, 4, 4, 4, 4> iGPvv_2 =
325:           iG_Pvv(lts.pfinal[0], lts.q2, res.g_Tensor[0], res.g_Tensor[1]);
326:
327:       // Vector-Pseudoscalar-Pseudoscalar coupling
328:       const Tensor1<std::complex<double>, 4> iGvpsps =
329:           iG_vpsps(p3, p4, M0, res.hel.g_decay_tensor[0]);
330:
331:       // Two helicity states for incoming and outgoing protons
332:       std::size_t index = 0;
```



```
333:       FOR_PP_HELICITY;
334:
335:       // Apply proton leg helicity conservation / No helicity flip (high energy limit)
336:       if (ha != h1 || hb != h2) { continue; }
337:
338:       // Full proton-gamma-proton spinor structure (upper and lower vertex)
339:       const Tensor1<std::complex<double>, 4> iG_1 = iG_ypp(p1, pa, ubar_1[h1], u_a[ha]);
340:       const Tensor1<std::complex<double>, 4> iG_2 = iG_ypp(p2, pb, ubar_2[h2], u_b[hb]);
341:
342:       // Full proton-Pomeron-proton spinor structure (upper and lower vertex)
343:       const Tensor2<std::complex<double>, 4, 4> iG_1a = iG_Ppp(p1, pa, ubar_1[h1], u_a[ha]);
344:       const Tensor2<std::complex<double>, 4, 4> iG_2b = iG_Ppp(p2, pb, ubar_2[h2], u_b[hb]);
345:
346:       // Gamma[1] - Pomeron[2] amplitude
347:       const std::complex<double> iM_yP = iG_1(mu1) * iDy_1(mu1, mu2) * iGyV(mu2, nu1) *
348:                                          iDV_1(nu1, rho1) * iDV(rho2, nu2) * iGvpsps(nu2) *
349:                                          iGPvv_1(rho2, rho1, alpha1, beta1) *
350:                                          iDP_2(alpha1, beta1, alpha2, beta2) * iG_2b(alpha2, beta2);
351:
352:       // Pomeron[2] - Gamma[1] amplitude
353:       const std::complex<double> iM_Py = iG_2(mu1) * iDy_2(mu1, mu2) * iGyV(mu2, nu1) *
354:                                          iDV_2(nu1, rho1) * iDV(rho2, nu2) * iGvpsps(nu2) *
355:                                          iGPvv_2(rho2, rho1, alpha1, beta1) *
356:                                          iDP_1(alpha1, beta1, alpha2, beta2) * iG_1a(alpha2, beta2);
357:
358:       // Total helicity amplitude
359:       const std::complex<double> A = (-zi) * (iM_yP + iM_Py);
360:       lts.hamp[index] += A;  // Note +=
361:       ++index;
362:
363:       FOR_PP_HELICITY_END;
364:       maxindex = index > maxindex ? index : maxindex;
365:
366:       // ==================================================================
367:       // Pomeron-Pomeron-Pseudoscalar structure
368:       //
369:       //
370:     } else if (J == 0 && P == -1) {
371:       Tensor4<std::complex<double>, 4, 4, 4, 4> cvtx;
372:       std::vector<std::complex<double>>          eps3eps4;
373:       cvtx = iG_PPS_total(lts.q1, lts.q2, M0, "pseudoscalar", res.g_Tensor);
374:
375:       // Scalar BW-propagator
376:       const std::complex<double> iD = iD_MES(lts.pfinal[0], M0, Gamma);
377:
378:       // 2 x Gamma decay part
379:       if (lts.decaytree.size() != 2 || lts.decaytree[0].p.pdg != 22 ||
380:           lts.decaytree[1].p.pdg != 22) {
381:         throw std::invalid_argument(
382:             "MTensorPomeron:: [PS decay] Only decay to gamma pair supported");
383:       }
384:
385:       // Decay vertex with different outgoing helicity combinations
386:       Tensor2<std::complex<double>, 4, 4> iDECAY = iG_psvv(p3, p4, M0, res.hel.g_decay_tensor[0]);
387:       eps3eps4                                   = MasslessSpin1Pol1Sum(iDECAY, p3, p4);
388:
389:       // Two helicity states for incoming and outgoing protons
390:       std::size_t index = 0;
391:       FOR_PP_HELICITY;
392:
393:       // Apply proton leg helicity conservation / No helicity flip (high energy limit)
394:       if (ha != h1 || hb != h2) { continue; }
395:
396:       // Full proton-Pomeron-proton spinor structure (upper and lower vertex)
397:       // const Tensor2<std::complex<double>,4,4> iG_1a = iG_Ppp(p1, pa, ubar_1[h1], u_a[ha]);
398:       // const Tensor2<std::complex<double>,4,4> iG_2b = iG_Ppp(p2, pb, ubar_2[h2], u_b[hb]);
399:
400:       // Evaluate block
401:       const std::complex<double> block = (-zi) * iG_1a(mu1, nu1) * iDP_1(mu1, nu1, alpha1, beta1) *
402:                                          cvtx(alpha1, beta1, alpha2, beta2) * iD *
403:                                          iDP_2(alpha2, beta2, mu2, nu2) * iG_2b(mu2, nu2);
404:
405:       // s-channel amplitude for each outgoing vector state helicity combination
406:       for (const auto &ind : indices(eps3eps4)) {
407:         const std::complex<double> A = block * eps3eps4[ind];
408:         lts.hamp[index] += A;  // Note +=
409:         ++index;
410:       }
411:
412:       FOR_PP_HELICITY_END;
413:       maxindex = index > maxindex ? index : maxindex;
414:
415:       // ==================================================================
```



```
416:       // Pomeron-Pomeron-Tensor structure
417:       //
418:       //
419:     } else if (J == 2) {
420:       // Rank-6 tensor structure
421:       const MTensor<std::complex<double>> iGPPf2 = iG_PPT_total(lts.q1, lts.q2, M0, res.g_Tensor);
422:
423:       // Tensor BW-propagator
424:       const bool                                INDEX_UP = true;
425:       const Tensor4<std::complex<double>, 4, 4, 4, 4> iDf2 =
426:           iD_TMES(lts.pfinal[0], M0, Gamma, INDEX_UP);
427:
428:       // Total block
429:       std::vector<Tensor2<std::complex<double>, 4, 4>> iD;
430:
431:       // [PS PS] Pseudoscalar pair decay
432:       if (lts.decaytree[0].p.spinX2 == 0 && lts.decaytree[1].p.spinX2 == 0) {
433:         const Tensor2<std::complex<double>, 4, 4> iGf2psps =
434:             iG_f2psps(p3, p4, M0, res.hel.g_decay_tensor[0]);
435:
436:         Tensor2<std::complex<double>, 4, 4> temp;
437:         temp(mu1, nu1) = iDf2(mu1, nu1, rho1, rho2) * iGf2psps(rho1, rho2);
438:         iD.push_back(temp);
439:
440:         // [V  V] Massive vector pair decay
441:       } else if (lts.decaytree[0].p.spinX2 == 2 && lts.decaytree[1].p.spinX2 == 2 &&
442:                  lts.decaytree[0].p.pdg != 22 && lts.decaytree[1].p.mass != 22) {
443:         if (res.hel.g_decay_tensor.size() != 2) {
444:           throw std::invalid_argument(
445:               "MTensorPomeron:: T->VV Coupling array [size 2] 'hel.g_decay_tensor' not in "
446:               "BRANCHING.json for resonance PDG = " +
447:               std::to_string(res.p.pdg));
448:         }
449:         const Tensor4<std::complex<double>, 4, 4, 4, 4> iGf2vv =
450:             iG_f2vv(p3, p4, M0, res.hel.g_decay_tensor[0], res.hel.g_decay_tensor[1]);
451:         const std::vector<Tensor2<std::complex<double>, 4, 4>> eps3eps4 =
452:             MassiveSpin1Pol1Sum(iGf2vv, p3, p4);
453:
454:         // Over all helicity combinations
455:         Tensor2<std::complex<double>, 4, 4> temp;
456:         for (const auto &ind : indices(eps3eps4)) {
457:           // Contract
458:           temp(mu1, nu1) = iDf2(mu1, nu1, rho1, rho2) * eps3eps4[ind](rho1, rho2);
459:           iD.push_back(temp);
460:         }
461:
462:         // Sequential spin correlated decay treatment [TBD]
463:         /*
464:         const Tensor2<std::complex<double>, 4, 4> iGvv2psps = iG_vv2psps({}, lts.decaytree[0].p.pdg);
465:
466:         // Contract in two steps
467:         Tensor2<std::complex<double>, 4, 4> A;
468:         A(alpha1, beta1) = iGPvv(rho1, rho2, alpha1, beta1) * iGvv2psps(rho1, rho2);
469:         iD(mu1, nu1) = iDf2(mu1, nu1, alpha1, beta1) * A(alpha1, beta1);
470:
471:         // ------------------------------------------------------------------
472:         // *** CONTROL CASCADE SAMPLING ***
473:         lts.FORCE_FLATMASS2 = true;
474:         lts.FORCE_OFFSHELL  = 3.0;
475:
476:         lts.decaytree[0].PS_active = true;
477:         lts.decaytree[1].PS_active = true;
478:         // ------------------------------------------------------------------
479:         */
480:
481:         // [y  y] Gamma pair decay
482:       } else if (lts.decaytree[0].p.pdg == 22 && lts.decaytree[1].p.pdg == 22) {
483:         if (res.hel.g_decay_tensor.size() != 2) {
484:           throw std::invalid_argument(
485:               "MTensorPomeron:: T->VV Coupling array [size 2] 'hel.g_decay_tensor' not in "
486:               "BRANCHING.json for resonance PDG = " +
487:               std::to_string(res.p.pdg));
488:         }
489:         Tensor4<std::complex<double>, 4, 4, 4, 4> iGf2yy =
490:             iG_f2yy(p3, p4, M0, res.hel.g_decay_tensor[0], res.hel.g_decay_tensor[1]);
491:         const std::vector<Tensor2<std::complex<double>, 4, 4>> eps3eps4 =
492:             MasslessSpin1Pol1Sum(iGf2yy, p3, p4);
493:
494:         // Over all helicity combinations
495:         Tensor2<std::complex<double>, 4, 4> temp;
496:         for (const auto &ind : indices(eps3eps4)) {
497:           // Contract
498:           temp(mu1, nu1) = iDf2(mu1, nu1, rho1, rho2) * eps3eps4[ind](rho1, rho2);
```

```
499:           iD.push_back(temp);
500:         }
501:
502:       } else {
503:         throw std::invalid_argument("MTensorPomeron::ME3: Unknown decay mode for tensor resonance");
504:       }
505:
506:       // Over all central helicity combinations
507:       Tensor4<std::complex<double>, 4, 4, 4, 4>                temp;
508:       std::vector<Tensor4<std::complex<double>, 4, 4, 4, 4>> cvtx;
509:       for (const auto &ind : indices[iD]) {
510:         // Construct central vertex
511:         FOR_EACH_4(LI);
512:         temp(u, v, k, l) = 0.0;
513:         for (auto const &rho : LI) {
514:           for (auto const &sigma : LI) {
515:             temp(u, v, k, l) += iGPPf2[{u, v, k, l, rho, sigma}] * iD[ind](rho, sigma);
516:           }
517:         }
518:         FOR_EACH_4_END;
519:         cvtx.push_back(temp);
520:       }
521:
522:       // Two helicity states for incoming and outgoing protons
523:       std::size_t index = 0;
524:       FOR_PP_HELICITY;
525:
526:       // Apply proton leg helicity conservation / No helicity flip (high energy limit)
527:       if (ha != h1 || hb != h2) { continue; }
528:
529:       // Full proton-Pomeron-proton spinor structure (upper and lower vertex)
530:       // const Tensor2<std::complex<double>,4,4> iG_1a = iG_Ppp(p1, pa, ubar_1[h1], u_a[ha]);
531:       // const Tensor2<std::complex<double>,4,4> iG_2b = iG_Ppp(p2, pb, ubar_2[h2], u_b[hb]);
532:
533:       // s-channel amplitude
534:       for (const auto &ind : indices(cvtx)) {
535:         const std::complex<double> A = (-zi) * iG_1a(mu1, nu1) * iDP_1(mu1, nu1, alpha1, beta1) *
536:                                        cvtx[ind](alpha1, beta1, alpha2, beta2) *
537:                                        iDP_2(alpha2, beta2, mu2, nu2) * iG_2b(mu2, nu2);
538:         // Add it
539:         lts.hamp[index] += A;  // Note +=
540:         ++index;
541:       }
542:
543:       FOR_PP_HELICITY_END;
544:       maxindex = index > maxindex ? index : maxindex;
545:
546:     } else {
547:       throw std::invalid_argument("MTensorPomeron::ME3: Unknown spin-parity input");
548:     }
549:     // ================================================================
550:
551:   }  // Loop over resonances
552:   // <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
553:
554:   // Remove empty
555:   lts.hamp.resize(maxindex);
556:
557:   // Get total amplitude squared 1/4 \sum_h |A_h|^2
558:   double SumAmp2 = 0.0;
559:   for (const auto &i : indices(lts.hamp)) { SumAmp2 += gra::math::abs2(lts.hamp[i]); }
560:   SumAmp2 /= 4;  // Initial state helicity average
561:
562:
563:   return SumAmp2;  // Amplitude squared
564: }
565:
566:
567: // 2 -> 4 amplitudes
568: //
569: // return value: matrix element squared with helicities summed over
570: // lts.hamp:     individual helicity amplitudes
571: //
572: double MTensorPomeron::ME4(gra::LORENTZSCALAR &lts) const {
573:   // Kinematics
574:   const M4Vec pa = lts.pbeam1;
575:   const M4Vec pb = lts.pbeam2;
576:
577:   const M4Vec p1 = lts.pfinal[1];
578:   const M4Vec p2 = lts.pfinal[2];
579:   M4Vec       p3 = lts.decaytree[0].p4;
580:   M4Vec       p4 = lts.decaytree[1].p4;
581:
```



```
582:   // Intermediate boson/fermion mass
583:   const double M_ = lts.decaytree[0].p.mass;
584:
585:   // Momentum convention of sub-diagrams
586:   //
587:   // ------<------ anti-particle p3
588:   // |
589:   //  \hat(t) (arrow down)
590:   // |
591:   // ------>------ particle p4
592:   //
593:   // ------<------ particle p4
594:   // |
595:   //  \hat(u) (arrow up)
596:   // |
597:   // ------>------ anti-particle p3
598:   //
599:   const M4Vec pt = pa - p1 - p3;  // => q1 = pt + p3, q2 = pt - p4
600:   const M4Vec pu = p4 - pa + p1;  // => q2 = pu + p3, q1 = pu - p3
601:
602:   // ================================================================
603:
604:   // Incoming and outgoing proton spinors (2 helicities)
605:   const std::array<std::vector<std::complex<double>>, 2> u_a = SpinorStates(pa, "u");
606:   const std::array<std::vector<std::complex<double>>, 2> u_b = SpinorStates(pb, "u");
607:
608:   const std::array<std::vector<std::complex<double>>, 2> ubar_1 = SpinorStates(p1, "ubar");
609:   const std::array<std::vector<std::complex<double>>, 2> ubar_2 = SpinorStates(p2, "ubar");
610:
611:
612:   // t-channel Pomeron propagators
613:   const Tensor4<std::complex<double>, 4, 4, 4, 4> iDP_13 = iD_P(lts.ss[1][3], lts.t1);
614:   const Tensor4<std::complex<double>, 4, 4, 4, 4> iDP_24 = iD_P(lts.ss[2][4], lts.t2);
615:
616:   // u-channel Pomeron propagators
617:   const Tensor4<std::complex<double>, 4, 4, 4, 4> iDP_14 = iD_P(lts.ss[1][4], lts.t1);
618:   const Tensor4<std::complex<double>, 4, 4, 4, 4> iDP_23 = iD_P(lts.ss[2][3], lts.t2);
619:
620:
621:   // High Energy Limit proton-Pomeron-proton spinor structure
622:   const Tensor2<std::complex<double>, 4, 4> iG_1a = iG_PppHE(p1, pa);
623:   const Tensor2<std::complex<double>, 4, 4> iG_2b = iG_PppHE(p2, pb);
624:
625:   // ================================================================
626:
627:   // Reset
628:   lts.hamp.clear();
629:
630:   // Free Lorentz indices [second parameter denotes the range of index]
631:   FTensor::Index<'a', 4> mu1;
632:   FTensor::Index<'b', 4> nu1;
633:   FTensor::Index<'c', 4> rho1;
634:   FTensor::Index<'d', 4> rho2;
635:   FTensor::Index<'g', 4> alpha1;
636:   FTensor::Index<'h', 4> beta1;
637:   FTensor::Index<'i', 4> alpha2;
638:   FTensor::Index<'j', 4> beta2;
639:   FTensor::Index<'k', 4> mu2;
640:   FTensor::Index<'l', 4> nu2;
641:
642:   // DEDUCE subprocess
643:   std::string SPINMODE;
644:
645:   if (lts.decaytree[0].p.spinX2 == 0 && lts.decaytree[1].p.spinX2 == 0) {
646:     SPINMODE = "2x5";
647:   } else if (lts.decaytree[0].p.spinX2 == 1 && lts.decaytree[1].p.spinX2 == 1) {
648:     SPINMODE = "2xV";
649:   } else if (lts.decaytree[0].p.spinX2 == 2 && lts.decaytree[1].p.spinX2 == 2) {
650:     SPINMODE = "2xV";
651:   } else {
652:     throw std::invalid_argument(
653:         "MTensorPomeron::ME4: Invalid daughter spin (J = 0, 1/2, 1 pairs supported)");
654:   }
655:
656:
657:   // Two helicity states for incoming and outgoing protons
658:   FOR_PP_HELICITY;
659:   // Apply proton leg helicity conservation / No helicity flip
660:   // (high energy limit)
661:   // This gives at least 4 x speed improvement
662:   if (ha != h1 || hb != h2) { continue; }
663:
664:   // ================================================================
```



```
665:   // Lepton or quark pair via two photon fusion
666:   if (SPINMODE == "2x5" && std::abs(lts.decaytree[0].p.pdg) <= 15) {
667:     // Full proton-gamma-proton spinor structure (upper and lower vertex)
668:     const Tensor1<std::complex<double>, 4> iG_1 = iG_ypp(p1, pa, ubar_1[h1], u_a[ha]);
669:     const Tensor1<std::complex<double>, 4> iG_2 = iG_ypp(p2, pb, ubar_2[h2], u_b[hb]);
670:
671:     // Photon propagators
672:     const Tensor2<std::complex<double>, 4, 4> iD_1 = iD_y(lts.t1);
673:     const Tensor2<std::complex<double>, 4, 4> iD_2 = iD_y(lts.t2);
674:
675:     // Fermion propagator
676:     const MMatrix<std::complex<double>> iSF_t = iD_F(pt, M_);
677:     const MMatrix<std::complex<double>> iSF_u = iD_F(pu, M_);
678:
679:     // Central spinors (2 helicities)
680:     const std::array<std::vector<std::complex<double>>, 2> v_3    = SpinorStates(p3, "v");
681:     const std::array<std::vector<std::complex<double>>, 2> ubar_4 = SpinorStates(p4, "ubar");
682:
683:
684:     double FACTOR = 1.0;
685:     // ----------------------------------------------------------------
686:     // quark pair (charge 1/3 or 2/3), apply charge and color factors
687:     if (std::abs(lts.decaytree[0].p.pdg) <= 6) {  // we have a quark
688:
689:       const double Q  = lts.decaytree[0].p.chargeX3 / 3.0;
690:       const double NC = 3.0;                    // quarks come in three colors
691:       FACTOR          = msqrt(math::pow4(Q) * NC);  // sqrt to "amplitude level"
692:     }
693:     // ----------------------------------------------------------------
694:
695:     for (const auto &h3 : indices(v_3)) {
696:       for (const auto &h4 : indices(ubar_4)) {
697:         // Vertex functions
698:         const Tensor2<std::complex<double>, 4, 4> iyFy_t = iG_yeebary(ubar_4[h4], iSF_t, v_3[h3]);
699:         const Tensor2<std::complex<double>, 4, 4> iyFy_u = iG_yeebary(ubar_4[h4], iSF_u, v_3[h3]);
700:
701:         // Full amplitude: t-channel + u_channel
702:         std::complex<double> M_tu = (-zi) * iG_1(mu1) * iD_1(mu1, nu1) *
703:                                     (iyFy_t(nu2, nu1) + iyFy_u(nu1, nu2)) * iD_2(nu2, mu2) *
704:                                     iG_2(mu2);
705:
706:         M_tu *= FACTOR;
707:
708:         lts.hamp.push_back(M_tu);
709:       }
710:     }
711:     continue;  // skip parts below, only for Pomeron amplitudes
712:   }
713:
714:   // ----------------------------------------------------------------
715:   // ----------------------------------------------------------------
716:   // Full proton-Pomeron-proton spinor structure (upper and lower vertex)
717:   /*
718:   const Tensor2<std::complex<double>,4,4> iG_1a = iG_Ppp(p1, pa, ubar_1[h1], u_a[ha]);
719:   const Tensor2<std::complex<double>,4,4> iG_2b = iG_Ppp(p2, pb, ubar_2[h2], u_b[hb]);
720:   */
721:
722:   // ================================================================
723:   // 2 x pseudoscalar (pion pair, kaon pair ...)
724:   if (SPINMODE == "2x5") {
725:     double gP = 0.0;
726:
727:     // Pion or Kaon coupling
728:     gP = (lts.decaytree[0].p.mass < 0.2) ? Param.gPpipi : Param.gPKK;
729:
730:     // t-channel blocks
731:     const Tensor2<std::complex<double>, 4, 4> iG_ta   = iG_Ppsps(pt, -p3, gP);
732:     const std::complex<double>               iDMES_t = iD_MES0(pt, M_);
733:     const Tensor2<std::complex<double>, 4, 4> iG_tb   = iG_Ppsps(p4, pt, gP);
734:
735:     // u-channel blocks
736:     const Tensor2<std::complex<double>, 4, 4> iG_ua   = iG_Ppsps(p4, pu, gP);
737:     const std::complex<double>               iDMES_u = iD_MES0(pu, M_);
738:     const Tensor2<std::complex<double>, 4, 4> iG_ub   = iG_Ppsps(pu, -p3, gP);
739:
740:     std::complex<double> M_t;
741:     std::complex<double> M_u;
742:     // t-channel
743:     {
744:       M_t = iG_1a(mu1, nu1) * iDP_13(mu1, nu1, alpha1, beta1) * iG_ta(alpha1, beta1) *
745:             iG_tb(alpha2, beta2) * iDP_24(alpha2, beta2, mu2, nu2) * iG_2b(mu2, nu2);
746:
747:
```

```
748:       // Apply propagator and off-shell form factors (two of them)
749:       M_t *= iDMES_t * pow2(PARAM_REGGE::Meson_FF(pt.M2(), pow2(M_)));
750:     }
751:
752:     // u-channel
753:     {
754:       M_u = iG_1a(mu1, nu1) * iDP_14(mu1, nu1, alpha1, beta1) * iG_ua(alpha1, beta1) *
755:             iG_ub(alpha2, beta2) * iDP_23(alpha2, beta2, mu2, nu2) * iG_2b(mu2, nu2);
756:       // Apply propagator and off-shell form factors (two of them)
757:       M_u *= iDMES_u * pow2(PARAM_REGGE::Meson_FF(pu.M2(), pow2(M_)));
758:     }
759:
760:     // Full amplitude: iM = [ ... ]  <-> M = (-i)*[ ... ]
761:     const std::complex<double> M = (-zi) * (M_t + M_u);
762:     lts.hamp.push_back(M);
763:   }
764:
765:   // ================================================================
766:   // 2 x fermion (proton-antiproton pair, lambda pair ...)
767:   else if (SPINMODE == "2xF") {
768:     // Fermion propagator
769:     const MMatrix<std::complex<double>> iSF_t = iD_F(pt, M_);
770:     const MMatrix<std::complex<double>> iSF_u = iD_F(pu, M_);
771:
772:     // Central spinors (2 helicities)
773:     const std::array<std::vector<std::complex<double>>, 2> v_3    = SpinorStates(p3, "v");
774:     const std::array<std::vector<std::complex<double>>, 2> ubar_4 = SpinorStates(p4, "ubar");
775:
776:     // Central fermion helicities
777:     for (const auto &h3 : indices(v_3)) {
778:       for (const auto &h4 : indices(ubar_4)) {
779:         // Fermion propagator and connected parts
780:         const Tensor4<std::complex<double>, 4, 4, 4, 4> iG_t =
781:             iG_PppbarP(p4, ubar_4[h4], pt, iSF_t, v_3[h3], -p3);
782:         const Tensor4<std::complex<double>, 4, 4, 4, 4> iG_u =
783:             iG_PppbarP(p4, ubar_4[h4], pu, iSF_u, v_3[h3], -p3);
784:
785:         // t-channel
786:         std::complex<double> M_t = iG_1a(mu1, nu1) * iDP_13(mu1, nu1, alpha1, beta1) *
787:                                    iG_t(alpha2, beta2, alpha1, beta1) *
788:                                    iDP_24(alpha2, beta2, mu2, nu2) * iG_2b(mu2, nu2);
789:         // Apply off-shell form factors (two of them)
790:         M_t *= pow2(PARAM_REGGE::Baryon_FF(pt.M2(), pow2(M_)));
791:
792:         // u-channel
793:         std::complex<double> M_u = iG_1a(mu1, nu1) * iDP_14(mu1, nu1, alpha1, beta1) *
794:                                    iG_u(alpha1, beta1, alpha2, beta2) *
795:                                    iDP_23(alpha2, beta2, mu2, nu2) * iG_2b(mu2, nu2);
796:         // Apply off-shell form factors (two of them)
797:         M_u *= pow2(PARAM_REGGE::Baryon_FF(pu.M2(), pow2(M_)));
798:
799:         // Full amplitude: iM = [ ... ]  <-> M = (-i)*[ ... ]
800:         const std::complex<double> M = (-zi) * (M_t + M_u);
801:         lts.hamp.push_back(M);
802:       }
803:     }
804:   }
805:
806:   // ================================================================
807:   // 2 x vector meson (rho pair, phi pair ...)
808:   else if (SPINMODE == "2xV") {
809:     const int pdg = lts.decaytree[0].p.pdg;
810:
811:     double g1 = 0.0;
812:     double g2 = 0.0;
813:
814:     // Couplings (rho770 meson)
815:     if (pdg == 113) {
816:       g1 = Param.gPrhorho[0];
817:       g2 = Param.gPrhorho[1];
818:     }
819:     // Couplings (phi1020 meson)
820:     else if (pdg == 333) {
821:       g1 = Param.gPphiphi[0];
822:       g2 = Param.gPphiphi[1];
823:     }
824:
825:     FTensor::Index<'e', 4> rho3;
826:     FTensor::Index<'f', 4> rho4;
827:
828:     // t-channel blocks
829:     const Tensor4<std::complex<double>, 4, 4, 4, 4> iG_tA = iG_Pvv(pt, -p3, g1, g2);
830:     const Tensor2<std::complex<double>, 4, 4>       iDV_t = iD_V(pt, M_, lts.pfinal[0].M2());
```

```
831:     const Tensor4<std::complex<double>>, 4, 4, 4, 4> iG_tB = iG_Pvv(p4, pt, g1, g2);
832:
833:     // u-channel blocks
834:     const Tensor4<std::complex<double>>, 4, 4, 4, 4> iG_uA = iG_Pvv(p4, pu, g1, g2);
835:     const Tensor2<std::complex<double>>, 4, 4>       iDV_u = iD_V(pu, M_, lts.pfinal[0].M2());
836:     const Tensor4<std::complex<double>>, 4, 4, 4, 4> iG_uB = iG_Pvv(pu, -p3, g1, g2);
837:
838:     // ------------------------------------------------------------------
839:     // TENSOR LORENTZ INDEX CONTRACTION BLOCK
840:     // This is done in pieces, due to template <>
841:     // argument deduction constraints
842:
843:     Tensor2<std::complex<double>>, 4, 4> M_t;
844:     Tensor2<std::complex<double>>, 4, 4> M_u;
845:
846:     // t-channel
847:     {
848:       // Upper block
849:       Tensor2<std::complex<double>>, 4, 4> A;
850:       A(rho1, rho3) =
851:           iG_1a(mu1, nu1) * iDP_13(mu1, nu1, alpha1, beta1) * iG_tA(rho1, rho3, alpha1, beta1);
852:       // Lower block
853:       Tensor2<std::complex<double>>, 4, 4> B;
854:       B(rho4, rho2) =
855:           iG_2b(mu2, nu2) * iDP_24(alpha2, beta2, mu2, nu2) * iG_tB(rho4, rho2, alpha2, beta2);
856:       // Apply off-shell form factors (two of them)
857:       M_t(rho3, rho4) = A(rho1, rho3) * iDV_t(rho1, rho2) * B(rho4, rho2) *
858:                         pow2(PARAM_REGGE::Meson_FF(pt.M2(), pow2(M_)));
859:     }
860:
861:     // u-channel
862:     {
863:       // Upper block
864:       Tensor2<std::complex<double>>, 4, 4> A;
865:       A(rho4, rho1) =
866:           iG_1a(mu1, nu1) * iDP_14(mu1, nu1, alpha1, beta1) * iG_uA(rho4, rho1, alpha1, beta1);
867:       // Lower block
868:       Tensor2<std::complex<double>>, 4, 4> B;
869:       B(rho2, rho3) =
870:           iG_2b(mu2, nu2) * iDP_23(alpha2, beta2, mu2, nu2) * iG_uB(rho2, rho3, alpha2, beta2);
871:       // Apply off-shell form factors (two of them)
872:       M_u(rho3, rho4) = A(rho4, rho1) * iDV_u(rho1, rho2) * B(rho2, rho3) *
873:                         pow2(PARAM_REGGE::Meson_FF(pu.M2(), pow2(M_)));
874:     }
875:
876:     // Total amplitude: iM = [ ... ]  <-> M = (-i)*[ ... ]
877:     Tensor2<std::complex<double>>, 4, 4> M;
878:     FOR_EACH_2(LI);
879:     M(u, v) = (-zi) * (M_t(u, v) + M_u(u, v));
880:     FOR_EACH_2_END;
881:
882:     // Calculate helicity amplitudes
883:
884:     // Switch both Lorentz indices up
885:     // M(alphal, beta1) = M(mu1, nu1) * gT(mu1, alpha1) * gT(nu1, beta1);
886:     const std::vector<std::complex<double>> helamp = MassiveSpin1PolSum(M, p3, p4);
887:     lts.hamp.insert(lts.hamp.end(), helamp.begin(), helamp.end());
888:   }  // 2x1
889:
890:   FOR_PP_HELICITY_END;
891:
892:   // Get total amplitude squared 1/4 \sum_h |A_h|^2
893:   double SumAmp2 = 0.0;
894:   for (const auto &i : indices(lts.hamp)) { SumAmp2 += math::abs2(lts.hamp[i]); }
895:   SumAmp2 /= 4;  // Initial state helicity average
896:
897:   return SumAmp2;  // Amplitude squared
898: }
899:
900:
901: // [!!! THIS FUNCTION UNDER DEVELOPMENT >>>]
902: //
903: // 2 -> 6 amplitudes
904: //
905: // return value: matrix element squared with helicities summed over
906: // lts.hamp:     individual helicity amplitudes
907: //
908: double MTensorPomeron::ME6(gra::LORENTZSCALAR &lts) const {
909:   // Free Lorentz indices [second parameter denotes the range of index]
910:   FTensor::Index<'a', 4> mu1;
911:   FTensor::Index<'b', 4> nu1;
912:   FTensor::Index<'c', 4> rho1;
913:   FTensor::Index<'d', 4> rho2;
```



```
914:   FTensor::Index<'g', 4> alpha1;
915:   FTensor::Index<'h', 4> beta1;
916:   FTensor::Index<'i', 4> alpha2;
917:   FTensor::Index<'j', 4> beta2;
918:   FTensor::Index<'k', 4> mu2;
919:   FTensor::Index<'l', 4> nu2;
920:
921:   // Kinematics
922:   const M4Vec pa = lts.pbeam1;
923:   const M4Vec pb = lts.pbeam2;
924:
925:   const M4Vec p1 = lts.pfinal[1];
926:   const M4Vec p2 = lts.pfinal[2];
927:
928:   std::vector<M4Vec> pf(4);
929:
930:   // Intermediate boson/fermion mass
931:   double M_ = 0.0;
932:
933:   if (lts.decaytree.size() == 2) {
934:     pf[0] = lts.decaytree[0].legs[0].p4;
935:     pf[1] = lts.decaytree[0].legs[1].p4;
936:     pf[2] = lts.decaytree[1].legs[0].p4;
937:     pf[3] = lts.decaytree[1].legs[1].p4;
938:
939:     M_ = lts.decaytree[0].legs[0].p.mass;
940:   }
941:   if (lts.decaytree.size() == 4) {
942:     pf[0] = lts.decaytree[0].p4;
943:     pf[1] = lts.decaytree[1].p4;
944:     pf[2] = lts.decaytree[2].p4;
945:     pf[3] = lts.decaytree[3].p4;
946:
947:     M_ = lts.decaytree[0].p.mass;
948:   }
949:
950:   // ------------------------------------------------------------------
951:
952:   // Incoming and outgoing proton spinors (2 helicities)
953:   const std::array<std::vector<std::complex<double>>, 2> u_a = SpinorStates(pa, "u");
954:   const std::array<std::vector<std::complex<double>>, 2> u_b = SpinorStates(pb, "u");
955:
956:   const std::array<std::vector<std::complex<double>>, 2> ubar_1 = SpinorStates(p1, "ubar");
957:   const std::array<std::vector<std::complex<double>>, 2> ubar_2 = SpinorStates(p2, "ubar");
958:
959:   // ------------------------------------------------------------------
960:
961:   // Reset
962:   lts.hamp.clear();
963:
964:   // High Energy Limit proton-Pomeron-proton spinor structure
965:   const Tensor2<std::complex<double>>, 4, 4> iG_1a = iG_PppHE(p1, pa);
966:   const Tensor2<std::complex<double>>, 4, 4> iG_2b = iG_PppHE(p2, pb);
967:
968:   int PDG = 0;
969:   if (lts.decaytree.size() == 2) { PDG = lts.decaytree[0].p.pdg; }
970:   if (lts.decaytree.size() == 4) {
971:     // rho->pipi or phi->KK, based on daughters
972:     PDG = (lts.decaytree[0].p.mass) < 0.2 ? 113 : 333;
973:   }
974:
975:   double g1 = 0.0;
976:   double g2 = 0.0;
977:
978:   // Couplings (rho770 meson)
979:   if (PDG == 113) {
980:     g1 = Param.gPrhorho[0];
981:     g2 = Param.gPrhorho[1];
982:   }
983:   // Couplings (phi1020 meson)
984:   else if (PDG == 333) {
985:     g1 = Param.gPhiphi[0];
986:     g2 = Param.gPhiphi[1];
987:   }
988:
989:   FTensor::Index<'e', 4> rho3;
990:   FTensor::Index<'f', 4> rho4;
991:
992:   // Two helicity states for incoming and outgoing protons
993:   FOR_PP_HELICITY;
994:
995:   // Apply proton leg helicity conservation / No helicity flip (high energy limit)
996:   // This gives at least 4 x speed improvement
```



```
997:   if (ha != h1 || hb != h2) { continue; }
998:
999:   std::complex<double> amp = 0;
1000:
1001:   // Different permutations
1002:   const MMatrix<int> R = {{0, 1, 2, 3}, {0, 2, 1, 3}, {0, 3, 2, 1}, {2, 1, 0, 3}};
1003:
1004:   for (std::size_t pind = 0; pind < 1; ++pind) {
1005:     // ------------------------------------------------------------------
1006:
1007:     // Full proton-Pomeron-proton spinor structure (upper and lower vertex)
1008:     /*
1009:     const Tensor2<std::complex<double>,4,4> iG_1a = iG_Ppp(p1, pa, ubar_1[h1], u_a[ha]);
1010:     const Tensor2<std::complex<double>,4,4> iG_2b = iG_Ppp(p2, pb, ubar_2[h2], u_b[hb]);
1011:     */
1012:
1013:     const M4Vec p3 = pf[R[pind][0]] + pf[R[pind][1]];
1014:     const M4Vec p4 = pf[R[pind][2]] + pf[R[pind][3]];
1015:
1016:     // Momentum convention of sub-diagrams
1017:     //
1018:     // ------<------ anti-particle p3
1019:     // |
1020:     // |  \hat(t) (arrow down)
1021:     // |
1022:     // ------>------ particle p4
1023:     //
1024:     // ------<------ particle p4
1025:     // |
1026:     // |  \hat(u) (arrow up)
1027:     // |
1028:     // ------>------ anti-particle p3
1029:     //
1030:     const M4Vec pt = pa - p1 - p3;  // => q1 = pt + p3, q2 = pt - p4
1031:     const M4Vec pu = p4 - pa + p1;  // => q2 = pu + p3, q1 = pu - p3
1032:
1033:
1034:     const double s13 = (lts.pfinal[1] + p3).M2();
1035:     const double s24 = (lts.pfinal[2] + p4).M2();
1036:     const double s14 = (lts.pfinal[1] + p4).M2();
1037:     const double s23 = (lts.pfinal[2] + p3).M2();
1038:
1039:
1040:     // t-channel Pomeron propagators
1041:     const Tensor4<std::complex<double>>, 4, 4, 4, 4> iDP_13 = iD_P(s13, lts.t1);
1042:     const Tensor4<std::complex<double>>, 4, 4, 4, 4> iDP_24 = iD_P(s24, lts.t2);
1043:
1044:     // u-channel Pomeron propagators
1045:     const Tensor4<std::complex<double>>, 4, 4, 4, 4> iDP_14 = iD_P(s14, lts.t1);
1046:     const Tensor4<std::complex<double>>, 4, 4, 4, 4> iDP_23 = iD_P(s23, lts.t2);
1047:
1048:
1049:     // t-channel blocks
1050:     const Tensor4<std::complex<double>>, 4, 4, 4, 4> iG_tA = iG_Pvv(pt, -p3, g1, g2);
1051:     const Tensor2<std::complex<double>>, 4, 4>       iDV_t = iD_V(pt, M_, lts.pfinal[0].M2());
1052:     const Tensor4<std::complex<double>>, 4, 4, 4, 4> iG_tB = iG_Pvv(p4, pt, g1, g2);
1053:
1054:     // u-channel blocks
1055:     const Tensor4<std::complex<double>>, 4, 4, 4, 4> iG_uA = iG_Pvv(p4, pu, g1, g2);
1056:     const Tensor2<std::complex<double>>, 4, 4>       iDV_u = iD_V(pu, M_, lts.pfinal[0].M2());
1057:     const Tensor4<std::complex<double>>, 4, 4, 4, 4> iG_uB = iG_Pvv(pu, -p3, g1, g2);
1058:
1059:
1060:     // ------------------------------------------------------------------
1061:     // TENSOR LORENTZ INDEX CONTRACTION BLOCK
1062:     // This is done in pieces, due to template <>
1063:     // argument deduction constraints
1064:     Tensor2<std::complex<double>>, 4, 4> M_t;
1065:     Tensor2<std::complex<double>>, 4, 4> M_u;
1066:
1067:     // t-channel
1068:     {
1069:       // Upper block
1070:       Tensor2<std::complex<double>>, 4, 4> A;
1071:       A(rho1, rho3) =
1072:           iG_1a(mu1, nu1) * iDP_13(mu1, nu1, alpha1, beta1) * iG_tA(rho1, rho3, alpha1, beta1);
1073:       // Lower block
1074:       Tensor2<std::complex<double>>, 4, 4> B;
1075:       B(rho4, rho2) =
1076:           iG_2b(mu2, nu2) * iDP_24(alpha2, beta2, mu2, nu2) * iG_tB(rho4, rho2, alpha2, beta2);
1077:       // Apply off-shell form factors (two of them)
1078:       M_t(rho3, rho4) = A(rho1, rho3) * iDV_t(rho1, rho2) * B(rho4, rho2) *
1079:                         pow2(PARAM_REGGE::Meson_FF(pt.M2(), pow2(M_)));
```



```
1080:     }
1081:
1082:     // u-channel
1083:     {
1084:       // Upper block
1085:       Tensor2<std::complex<double>>, 4, 4> A;
1086:       A(rho4, rho1) =
1087:           iG_1a(mu1, nu1) * iDP_14(mu1, nu1, alpha1, beta1) * iG_uA(rho4, rho1, alpha1, beta1);
1088:       // Lower block
1089:       Tensor2<std::complex<double>>, 4, 4> B;
1090:       B(rho2, rho3) =
1091:           iG_2b(mu2, nu2) * iDP_23(alpha2, beta2, mu2, nu2) * iG_uB(rho2, rho3, alpha2, beta2);
1092:       // Apply off-shell form factors (two of them)
1093:       M_u(rho3, rho4) = A(rho4, rho1) * iDV_u(rho1, rho2) * B(rho2, rho3) *
1094:                         pow2(PARAM_REGGE::Meson_FF(pu.M2(), pow2(M_)));
1095:     }
1096:
1097:     // Vector -> PS Decay block
1098:     Tensor2<std::complex<double>>, 4, 4> DECAY =
1099:         iG_vv2psps(pf[R[pind][0]], pf[R[pind][1]], pf[R[pind][2]], pf[R[pind][3]]}, PDG);
1100:
1101:     // Add
1102:     amp += (-zi) * (M_t(rho3, rho4) + M_u(rho3, rho4)) * DECAY(rho3, rho4);
1103:
1104:   }  // Permutations
1105:
1106:   lts.hamp.push_back(amp);
1107:   FOR_PP_HELICITY_END;
1108:
1109:   // Get total amplitude squared 1/4 \sum_h |A_h|^2
1110:   double SumAmp2 = 0.0;
1111:   for (const auto &i : indices(lts.hamp)) { SumAmp2 += math::abs2(lts.hamp[i]); }
1112:   SumAmp2 /= 4;  // Initial state helicity average
1113:
1114:   // ------------------------------------------------------------------
1115:   // *** CONTROL CASCADE SAMPLING ***
1116:   lts.FORCE_FLATMASS2 = true;
1117:   lts.FORCE_OFFSHELL  = 3.0;
1118:
1119:   lts.decaytree[0].PS_active = true;
1120:   lts.decaytree[1].PS_active = true;
1121:   // ------------------------------------------------------------------
1122:
1123:   return SumAmp2;  // Amplitude squared
1124: }
1125: // [<<< THIS FUNCTION UNDER DEVELOPMENT !!!]
1126:
1127:
1128: // External massless spin-1 polarization sum for outgoing states
1129: //
1130: // Input tensor M_[\mu \nu \kappa \rho] (indices down)
1131: //
1132: std::vector<Tensor2<std::complex<double>>, 4, 4>> MTensorPomeron::MasslessSpin1PolSum(
1133:     const Tensor4<std::complex<double>>, 4, 4, 4, 4> &M, const M4Vec &p3, const M4Vec &p4) const {
1134:   FTensor::Index<'a', 4> rho3;
1135:   FTensor::Index<'b', 4> rho4;
1136:   FTensor::Index<'c', 4> mu3;
1137:   FTensor::Index<'d', 4> mu4;
1138:
1139:   std::vector<Tensor2<std::complex<double>>, 4, 4>> hamp;
1140:   Tensor2<std::complex<double>>, 4, 4>               temp;
1141:
1142:   // Massless polarization vectors (2 helicities)
1143:   const bool                                         INDEX_UP = true;
1144:   const std::array<Tensor1<std::complex<double>>, 4>, 2> eps_3_conj =
1145:       MasslessSpin1States(p3, "conj", INDEX_UP);
1146:   const std::array<Tensor1<std::complex<double>>, 4>, 2> eps_4_conj =
1147:       MasslessSpin1States(p4, "conj", INDEX_UP);
1148:
1149:   // Loop over states
1150:   for (const auto &h3 : indices(eps_3_conj)) {
1151:     for (const auto &h4 : indices(eps_4_conj)) {
1152:       // Autocontraction
1153:       temp(mu3, mu4) = eps_3_conj[h3](rho3) * eps_4_conj[h4](rho4) * M(rho3, rho4, mu3, mu4);
1154:       hamp.push_back(temp);
1155:     }
1156:   }
1157:   return hamp;
1158: }
1159:
1160:
1161: // External massless spin-1 polarization sum for outgoing states
1162: //
```

```
1163: // Input tensor M_{\mu \nu} (indices down)
1164: //
1165: std::vector<std::complex<double>> MTensorPomeron::MasslessSpin1PolSum(
1166:     const Tensor2<std::complex<double>, 4, 4> &M, const M4Vec &p3, const M4Vec &p4) const {
1167:   FTensor::Index<'a', 4>              rho3;
1168:   FTensor::Index<'b', 4>              rho4;
1169:   std::vector<std::complex<double>> hamp;
1170:
1171:   // Massless polarization vectors (2 helicities)
1172:   const bool                                INDEX_UP = true;
1173:   const std::array<Tensor1<std::complex<double>, 4>, 2> eps_3_conj =
1174:       MasslessSpin1States(p3, "conj", INDEX_UP);
1175:   const std::array<Tensor1<std::complex<double>, 4>, 2> eps_4_conj =
1176:       MasslessSpin1States(p4, "conj", INDEX_UP);
1177:
1178:   // Loop over massive Spin-1 helicity states
1179:   for (const auto &h3 : indices(eps_3_conj)) {
1180:     for (const auto &h4 : indices(eps_4_conj)) {
1181:       // Autocontraction
1182:       const std::complex<double> amp = eps_3_conj[h3](rho3) * eps_4_conj[h4](rho4) * M(rho3, rho4);
1183:       hamp.push_back(amp);
1184:     }
1185:   }
1186:   return hamp;
1187: }
1188:
1189:
1190: // External massive spin-1 (vector) polarization sum for outgoing states
1191: //
1192: // Input tensor M_{\mu \nu \kappa \rho} (indices down)
1193: //
1194: std::vector<Tensor2<std::complex<double>, 4, 4>> MTensorPomeron::MassiveSpin1PolSum(
1195:     const Tensor4<std::complex<double>, 4, 4, 4, 4> &M, const M4Vec &p3, const M4Vec &p4) const {
1196:   FTensor::Index<'a', 4> rho3;
1197:   FTensor::Index<'b', 4> rho4;
1198:   FTensor::Index<'c', 4> mu3;
1199:   FTensor::Index<'d', 4> mu4;
1200:
1201:   std::vector<Tensor2<std::complex<double>, 4, 4>> hamp;
1202:   Tensor2<std::complex<double>, 4, 4>              temp;
1203:
1204:   // Massive polarization vectors (3 helicities)
1205:   const bool                                INDEX_UP = true;
1206:   const std::array<Tensor1<std::complex<double>, 4>, 3> eps_3_conj =
1207:       MassiveSpin1States(p3, "conj", INDEX_UP);
1208:   const std::array<Tensor1<std::complex<double>, 4>, 3> eps_4_conj =
1209:       MassiveSpin1States(p4, "conj", INDEX_UP);
1210:
1211:   // Loop over states
1212:   for (const auto &h3 : indices(eps_3_conj)) {
1213:     for (const auto &h4 : indices(eps_4_conj)) {
1214:       // Autocontraction
1215:       temp(mu3, mu4) = eps_3_conj[h3](rho3) * eps_4_conj[h4](rho4) * M(rho3, rho4, mu3, mu4);
1216:       hamp.push_back(temp);
1217:     }
1218:   }
1219:   return hamp;
1220: }
1221:
1222:
1223: // External massive spin-1 (vector) polarization sum for outgoing states
1224: //
1225: // Input tensor M_{\mu \nu \kappa \rho} (indices down)
1226: //
1227: std::vector<std::complex<double>> MTensorPomeron::MassiveSpin1PolSum(
1228:     const Tensor2<std::complex<double>, 4, 4> &M, const M4Vec &p3, const M4Vec &p4) const {
1229:   FTensor::Index<'a', 4>              rho3;
1230:   FTensor::Index<'b', 4>              rho4;
1231:   std::vector<std::complex<double>> hamp;
1232:
1233:   // Massive polarization vectors (3 helicities)
1234:   const bool                                INDEX_UP = true;
1235:   const std::array<Tensor1<std::complex<double>, 4>, 3> eps_3_conj =
1236:       MassiveSpin1States(p3, "conj", INDEX_UP);
1237:   const std::array<Tensor1<std::complex<double>, 4>, 3> eps_4_conj =
1238:       MassiveSpin1States(p4, "conj", INDEX_UP);
1239:
1240:   // Loop over massive Spin-1 helicity states
1241:   for (const auto &h3 : indices(eps_3_conj)) {
1242:     for (const auto &h4 : indices(eps_4_conj)) {
1243:       // Autocontraction
1244:       const std::complex<double> amp = eps_3_conj[h3](rho3) * eps_4_conj[h4](rho4) * M(rho3, rho4);
1245:       hamp.push_back(amp);
```



```
1246:     }
1247:   }
1248:
1249:   /*
1250:   // Polarization sum (completeness relation applied)
1251:   // already simplified expression => second terms vanishes and leaves only g[u][k] * g[v][l]
1252:   double Amp2 = 0.0;
1253:
1254:   FOR_EACH_4(LI);
1255:   const std::complex<double> contract = std::conj(M(u, v)) * M(k, l) * g[u][k] * g[v][l];
1256:   Amp2 += std::real(contract);      // real for casting to double
1257:   FOR_EACH_4_END;
1258:
1259:   hamp.push_back(msqrt(Amp2)); // Phase information lost here
1260:   */
1261:
1262:   return hamp;
1263: }
1264:
1265:
1266: // 2xvector -> 2xpseudoscalar decay block
1267: //
1268: Tensor2<std::complex<double>, 4, 4> MTensorPomeron::iG_vv2psps(const std::vector<M4Vec> &p,
1269:                                                                 int PDG) const {
1270:   FTensor::Index<'e', 4> rho3;
1271:   FTensor::Index<'f', 4> rho4;
1272:   FTensor::Index<'g', 4> kappa3;
1273:   FTensor::Index<'h', 4> kappa4;
1274:
1275:   double g1 = 0.0;
1276:   double M  = 0.0;
1277:   double W  = 0.0;
1278:   if (PDG == 113) {         // rho(770)0
1279:     const double BR = 1.0;  // pi+pi-
1280:     M               = 7.7526E-01;
1281:     W               = 1.491E-01;
1282:     g1              = GDecay(1, M, W, 1.3957061E-01, BR);
1283:   } else if (PDG == 333) {  // phi(1020)0
1284:     const double BR = 0.489;  // K+K-
1285:     M               = 1.019461E+00;
1286:     W               = 4.249E-03;
1287:     g1              = GDecay(1, M, W, 4.93677E-01, BR);
1288:   } else {
1289:     throw std::invalid_argument("MTensorPomeron::iG_vv2psps: Unsupported vector meson PDG = " +
1290:                                 std::to_string(PDG));
1291:   }
1292:
1293:   // Vector propagators
1294:   const bool                                INDEX_UP = true;
1295:   const Tensor2<std::complex<double>, 4, 4> iD_V1    = iD_VMES(p[0] + p[1], M, W, INDEX_UP);
1296:   const Tensor2<std::complex<double>, 4, 4> iD_V2    = iD_VMES(p[2] + p[3], M, W, INDEX_UP);
1297:
1298:   // Decay couplings
1299:   const Tensor1<std::complex<double>, 4> iG_D1 = iG_vpsps(p[0], p[1], M, g1);
1300:   const Tensor1<std::complex<double>, 4> iG_D2 = iG_vpsps(p[2], p[3], M, g1);
1301:
1302:   Tensor2<std::complex<double>, 4, 4> BLOCK;
1303:   BLOCK(rho3, rho4) = iD_V1(rho3, kappa3) * iG_D1(kappa3) * iD_V2(rho4, kappa4) * iG_D2(kappa4);
1304:
1305:   return BLOCK;
1306: }
1307:
1308:
1309: // ------------------------------------------------------------------------
1310: // Vertex functions
1311: //
1312: // Gamma-Lepton-Lepton vertex
1313: // contracted in \bar{spinor} G_\mu \bar{spinor}
1314: //
1315: // iGamma_\mu (p',p)
1316: //
1317: // High Energy Limit:
1318: // ubar(p',\lambda') \gamma_\mu u(p,\lambda ~= (p' + p)_\mu
1319: // \delta_{\lambda',\lambda}
1320: //
1321: // Input as contravariant (upper index) 4-vectors
1322: //
1323: Tensor1<std::complex<double>, 4> MTensorPomeron::iG_yee(
1324:     const M4Vec &prime, const M4Vec &p, const std::vector<std::complex<double>> &ubar,
1325:     const std::vector<std::complex<double>> &u) const {
1326:   // const double q2 = (prime-p).M2();
1327:   const double e = msqrt(qed::alpha_QED() * 4.0 * PI);  // ~ 0.3, no running
1328:
```



```
1329:   Tensor1<std::complex<double>, 4> T;
1330:   for (const auto &mu : LI) {
1331:     // \bar{spinor} [Gamma Matrix] \spinor product
1332:     T(mu) = zi * e * matoper::VecMatVecMultiply(ubar, gamma_lo[mu], u);
1333:   }
1334:   return T;
1335: }
1336:
1337: // Gamma-Proton-Proton vertex iGamma_\mu (p', p)
1338: // contracted in \bar{spinor} iG_\mu \bar{spinor}
1339: //
1340: // Input as contravariant (upper index) 4-vectors
1341: //
1342: Tensor1<std::complex<double>, 4> MTensorPomeron::iG_ypp(
1343:     const M4Vec &prime, const M4Vec &p, const std::vector<std::complex<double>> &ubar,
1344:     const std::vector<std::complex<double>> &u) const {
1345:   const double t    = (prime - p).M2();
1346:   const double e    = msqrt(qed::alpha_QED() * 4.0 * PI);  // ~ 0.3, no running
1347:   const M4Vec  psum = prime - p;
1348:
1349:   Tensor1<std::complex<double>, 4> T;
1350:   for (const auto &mu : LI) {
1351:     MMatrix<std::complex<double>> SUM(4, 4, 0.0);
1352:     for (const auto &nu : LI) { SUM += sigma_lo[mu][nu] * psum[nu]; }
1353:     const MMatrix<std::complex<double>> A = gamma_lo[mu] * F1(t);
1354:     const MMatrix<std::complex<double>> B = SUM * zi / (2 * PDG::mp) * F2(t);
1355:     const MMatrix<std::complex<double>> G = (A + B) * (-zi * e);
1356:
1357:     // \bar{spinor} [Gamma Matrix] \spinor product
1358:     T(mu) = matoper::VecMatVecMultiply(ubar, G, u);
1359:   }
1360:
1361:   return T;
1362: }
1363:
1364: // gamma - electron - fermion propagator - positron - gamma vertex
1365: // iGamma_{\mu \nu}
1366: //
1367: Tensor2<std::complex<double>, 4, 4> MTensorPomeron::iG_yeebary(
1368:     const std::vector<std::complex<double>> &ubar, const MMatrix<std::complex<double>> &iSF,
1369:     const std::vector<std::complex<double>> &v) const {
1370:   const double                e    = msqrt(qed::alpha_QED() * 4.0 * PI);  // ~ 0.3, no running
1371:   const std::complex<double> vertex = zi * e;
1372:
1373:   Tensor2<std::complex<double>, 4, 4> T;
1374:   for (const auto &mu : LI) {
1375:     const std::vector<std::complex<double>> lhs = matoper::VecMatMultiply(ubar, gamma_lo[mu] * iSF);
1376:
1377:     for (const auto &nu : LI) {
1378:       T(mu, nu) = vertex * matoper::VecMatVecMultiply(lhs, gamma_lo[nu], v) * vertex;
1379:     }
1380:   }
1381:   return T;
1382: }
1383:
1384:
1385: // High-Energy limit proton-Pomeron-proton vertex times \delta^{lambda_prime,
1386: // \lambda} (helicity conservation)
1387: // (~x 2 faster evaluation than the exact spinor structure)
1388: Tensor2<std::complex<double>, 4, 4> MTensorPomeron::iG_PppHE(const M4Vec &prime,
1389:                                                               const M4Vec  p) const {
1390:   const M4Vec                 psum   = prime + p;
1391:   const std::complex<double> FACTOR = -zi * 3.0 * Param.gPNN * F1((prime - p).M2());
1392:
1393:   Tensor2<std::complex<double>, 4, 4> T;
1394:   FOR_EACH_2(LI);
1395:   T(u, v) = FACTOR * (psum % u) * (psum % v);
1396:   FOR_EACH_2_END;
1397:
1398:   return T;
1399: }
1400:
1401: // Pomeron-Proton-Proton [-Neutron-Neutron, -Antiproton-Antiproton] vertex
1402: // contracted in \bar{spinor} G_{\mu\nu} \bar{spinor}
1403: //
1404: // i\Gamma_{\mu\nu} (p', p)
1405: //
1406: // Input as contravariant (upper index) 4-vectors
1407: //
1408: Tensor2<std::complex<double>, 4, 4> MTensorPomeron::iG_Ppp(
1409:     const M4Vec &prime, const M4Vec &p, const std::vector<std::complex<double>> &ubar,
1410:     const std::vector<std::complex<double>> &u) const {
1411:   const double t    = (prime - p).M2();
```



```
1412:   const M4Vec  psum = prime + p;
1413:
1414:   Tensor2<std::complex<double>, 4, 4> T;
1415:   const std::complex<double>          FACTOR = -zi * 3.0 * Param.gPNN * F1(t);
1416:
1417:   // Feynman slash
1418:   const MMatrix<std::complex<double>> slash = FSlash(psum);
1419:
1420:   // Aux matrix
1421:   MMatrix<std::complex<double>> A;
1422:
1423:   for (const auto &mu : LI) {
1424:     for (const auto &nu : LI) {
1425:       A = (gamma_lo[mu] * (psum % nu) + gamma_lo[nu] * (psum % mu)) * 0.5;
1426:       if (mu == nu) { A = A - slash * 0.25 * g[mu][nu]; }
1427:
1428:       // \bar{spinor} [Gamma Matrix] \spinor product
1429:       T(mu, nu) = FACTOR * gra::matoper::VecMatVecMultiply(ubar, A, u);
1430:     }
1431:   }
1432:   return T;
1433: }
1434:
1435: // Pomeron x outgoing proton spinor ubar x
1436: //           <antiproton/proton fermion propagator>
1437: //         x outgoing antiproton spinor v x Pomeron vertex function
1438: //
1439: // iG_{\mu_2\nu_2\mu_1\nu_1}
1440: //
1441: Tensor4<std::complex<double>, 4, 4, 4, 4> MTensorPomeron::iG_PppbarP(
1442:     const M4Vec &prime, const std::vector<std::complex<double>> &ubar, const M4Vec &pt,
1443:     const MMatrix<std::complex<double>> &iSF, const std::vector<std::complex<double>> &v,
1444:     const M4Vec &p1) const {
1445:   const std::complex<double> lhs_FACTOR = -zi * 3.0 * Param.gPNN * F1((prime - pt).M2());
1446:   const std::complex<double> rhs_FACTOR = -zi * 3.0 * Param.gPNN * F1((pt - p).M2());
1447:
1448:   const M4Vec lhs_psum = prime + pt;
1449:   const M4Vec rhs_psum = pt + p;
1450:
1451:   // Feynman slashes
1452:   const MMatrix<std::complex<double>> lhs_slash = FSlash(lhs_psum);
1453:   const MMatrix<std::complex<double>> rhs_slash = FSlash(rhs_psum);
1454:
1455:   // Aux matrices
1456:   MMatrix<std::complex<double>> lhs_A;
1457:   MMatrix<std::complex<double>> rhs_A;
1458:
1459:   Tensor4<std::complex<double>, 4, 4, 4, 4> T;
1460:
1461:   for (const auto &mu2 : LI) {
1462:     for (const auto &nu2 : LI) {
1463:       lhs_A = (gamma_lo[mu2] * (lhs_psum % nu2) + gamma_lo[nu2] * (lhs_psum % mu2)) * 0.5;
1464:       if (mu2 == nu2) { lhs_A = lhs_A - lhs_slash * 0.25 * g[mu2][nu2]; }
1465:
1466:       // Fermion propagator applied in the middle
1467:       const std::vector<std::complex<double>> ubarM = matoper::VecMatMultiply(ubar, lhs_A * iSF);
1468:
1469:       for (const auto &mu1 : LI) {
1470:         for (const auto &nu1 : LI) {
1471:           rhs_A = (gamma_lo[mu1] * (rhs_psum % nu1) + gamma_lo[nu1] * (rhs_psum % mu1)) * 0.5;
1472:           if (mu1 == nu1) { rhs_A = rhs_A - rhs_slash * 0.25 * g[mu1][nu1]; }
1473:
1474:           // \bar{spinor} [Gamma Matrix] \spinor product
1475:           T(mu2, nu2, mu1, nu1) =
1476:               lhs_FACTOR * matoper::VecMatVecMultiply(ubarM, rhs_A, v) * rhs_FACTOR;
1477:         }
1478:       }
1479:     }
1480:   }
1481:   return T;
1482: }
1483:
1484:
1485: // ========================================================================
1486:
1487: // Pomeron (\mu\nu) - Pomeron (\kappa\lambda) - Scalar/Pseudoscalar resonance
1488: // vertex function: i\Gamma_{\mu\nu,\kappa\lambda,\rho\sigma}
1489: //
1490: // Input as contravariant (upper index) 4-vector, M0 the resonance peak mass
1491: //
1492: Tensor4<std::complex<double>, 4, 4, 4, 4> MTensorPomeron::iG_PPS_total(
1493:     const M4Vec &q1, const M4Vec &q2, double M0, const std::string &mode,
1494:     const std::vector<double> &g_PPS) const {
```

```
1495:   // Form FACTOR
1496:   const double LAMBDA  = 1.0;  // GeV
1497:   const double F_tilde = FM(q1.M2(), 1.0) * FM(q2.M2(), 1.0) * F((q1 + q2).M2(), M0, LAMBDA);
1498:
1499:   // Tensor structures
1500:   Tensor4<std::complex<double>, 4, 4, 4, 4> T;
1501:
1502:   if (mode == "scalar") {
1503:     static const Tensor4<std::complex<double>, 4, 4, 4, 4> iG0 = MTensorPomeron::iG_PPS_0();
1504:     const Tensor4<std::complex<double>, 4, 4, 4, 4>        iG1 =
1505:         MTensorPomeron::iG_PPS_1(q1, q2, g_PPS[1]);
1506:
1507:     FOR_EACH_4(LI);
1508:     T(u, v, k, l) = (g_PPS[0] * iG0(u, v, k, l) + iG1(u, v, k, l)) * F_tilde;
1509:     FOR_EACH_4_END;
1510:
1511:   } else if (mode == "pseudoscalar") {
1512:     const Tensor4<std::complex<double>, 4, 4, 4, 4> iG0 =
1513:         MTensorPomeron::iG_PPPS_0(q1, q2, g_PPS[0]);
1514:     const Tensor4<std::complex<double>, 4, 4, 4, 4> iG1 =
1515:         MTensorPomeron::iG_PPPS_1(q1, q2, g_PPS[1]);
1516:
1517:     FOR_EACH_4(LI);
1518:     T(u, v, k, l) = (iG0(u, v, k, l) + iG1(u, v, k, l)) * F_tilde;
1519:     FOR_EACH_4_END;
1520:
1521:   } else {
1522:     throw std::invalid_argument(
1523:         "MTensorPomeron::iG_PPS_total: Unknown mode (should be scalar or pseudoscalar)");
1524:   }
1525:
1526:   return T;
1527: }
1528:
1529: // Pomeron-Pomeron-Scalar coupling structure #0
1530: // iG_{\mu \nu \kappa \lambda} ~ (1,s) = (0,0)
1531: //
1532: // Input as contravariant (upper index) 4-vectors
1533: //
1534: // Apply coupling g_PPS[0] outside this!
1535: //
1536: Tensor4<std::complex<double>, 4, 4, 4, 4> MTensorPomeron::iG_PPS_0() const {
1537:   const double               S0     = 1.0;  // Mass scale (GeV)
1538:   const std::complex<double> FACTOR = zi * S0;
1539:
1540:   Tensor4<std::complex<double>, 4, 4, 4, 4> T;
1541:   FOR_EACH_4(LI);
1542:   T(u, v, k, l) = FACTOR * (g[u][k] * g[v][l] + g[u][l] * g[v][k] - 0.5 * g[u][v] * g[k][l]);
1543:   FOR_EACH_4_END;
1544:
1545:   return T;
1546: }
1547:
1548: // Pomeron-Pomeron-Scalar coupling structure #1
1549: // iG_{\mu \nu \kappa \lambda} ~ (1,s) = (2,2)
1550: //
1551: // Input as contravariant (upper index) 4-vectors
1552: //
1553: Tensor4<std::complex<double>, 4, 4, 4, 4> MTensorPomeron::iG_PPS_1(const M4Vec &q1, const M4Vec &q2,
1554:                                                                    double g_PPS) const {
1555:   const double               S0     = 1.0;  // Mass scale (GeV)
1556:   const double               q1q2   = q1 * q2;
1557:   const std::complex<double> FACTOR = zi * g_PPS / (2 * S0);
1558:
1559:   Tensor4<std::complex<double>, 4, 4, 4, 4> T;
1560:   FOR_EACH_4(LI);
1561:   T(u, v, k, l) = FACTOR * ((q1 % k) * (q2 % u) * g[v][l] + (q1 % k) * (q2 % v) * g[u][l] +
1562:                             (q1 % l) * (q2 % u) * g[v][k] + (q1 % l) * (q2 % v) * g[u][k] -
1563:                             2.0 * q1q2 * (g[u][k] * g[v][l] + g[v][k] * g[u][l]));
1564:   FOR_EACH_4_END;
1565:
1566:   return T;
1567: }
1568:
1569: // Pomeron-Pomeron-Pseudoscalar coupling structure #0
1570: // iG_{\mu \nu \kappa \lambda} ~ (1,s) = (1,1)
1571: //
1572: // Input as contravariant (upper index) 4-vectors
1573: //
1574: Tensor4<std::complex<double>, 4, 4, 4, 4> MTensorPomeron::iG_PPPS_0(const M4Vec &q1,
1575:                                                                     const M4Vec &q2,
1576:                                                                     double       g_PPPS) const {
1577:   const double S0 = 1.0;  // Mass scale (GeV)
```



```
1578:
1579:   const M4Vec q1_q2 = q1 - q2;
1580:   const M4Vec q     = q1 + q2;
1581:
1582:   const std::complex<double>               FACTOR = zi * g_PPPS / (2.0 * S0);
1583:   Tensor4<std::complex<double>, 4, 4, 4, 4> T;
1584:
1585:   FOR_EACH_4(LI);
1586:   T(u, v, k, l) = 0.0;
1587:
1588:   // Contract r and s indices
1589:   for (const auto &r : LI) {
1590:     for (const auto &s : LI) {
1591:       // Note +=
1592:       T(u, v, k, l) += FACTOR *
1593:                        (g[u][k] * eps_lo(v, l, r, s) + g[v][k] * eps_lo(u, l, r, s) +
1594:                         g[u][l] * eps_lo(v, k, r, s) + g[v][l] * eps_lo(u, k, r, s)) *
1595:                        q1_q2[r] * q[s];
1596:     }
1597:   }
1598:   FOR_EACH_4_END;
1599:
1600:   return T;
1601: }
1602:
1603: // Pomeron-Pomeron-Pseudoscalar coupling structure #1
1604: // iG_{\mu \nu \kappa \lambda} ~ (1,s) = (3,3)
1605: //
1606: // Input as contravariant (upper index) 4-vectors
1607: //
1608: Tensor4<std::complex<double>, 4, 4, 4, 4> MTensorPomeron::iG_PPPS_1(const M4Vec &q1,
1609:                                                                     const M4Vec &q2,
1610:                                                                     double       g_PPPS) const {
1611:   const double               S0     = 1.0;  // Mass scale (GeV)
1612:   const M4Vec                q1_q2  = q1 - q2;
1613:   const M4Vec                q      = q1 + q2;
1614:   const double               q1q2   = q1 * q2;
1615:   const std::complex<double> FACTOR = zi * g_PPPS / gra::math::pow3(S0);
1616:
1617:   Tensor4<std::complex<double>, 4, 4, 4, 4> T;
1618:
1619:   FOR_EACH_4(LI);
1620:   T(u, v, k, l) = 0.0;
1621:
1622:   // Contract r and s indices
1623:   for (const auto &r : LI) {
1624:     for (const auto &s : LI) {
1625:       // Note +=
1626:       T(u, v, k, l) += FACTOR *
1627:                        (eps_lo(v, l, r, s) * ((q1 % k) * (q2 % u) - q1q2 * g[u][k]) +
1628:                         eps_lo(u, l, r, s) * ((q1 % k) * (q2 % v) - q1q2 * g[v][k]) +
1629:                         eps_lo(v, k, r, s) * ((q1 % l) * (q2 % u) - q1q2 * g[u][l]) +
1630:                         eps_lo(u, k, r, s) * ((q1 % l) * (q2 % v) - q1q2 * g[v][l])) *
1631:                        q1_q2[r] * q[s];
1632:     }
1633:   }
1634:   FOR_EACH_4_END;
1635:
1636:   return T;
1637: }
1638:
1639: // ============================================================
1640:
1641: // Pomeron (\mu\nu) - Pomeron (\kappa\lambda) - Tensor resonance (\rho\sigma)
1642: // vertex function: i\Gamma_{\mu\nu,\kappa\lambda,\rho\sigma}
1643: //
1644: // Input as contravariant (upper index) 4-vector, M0 the resonance peak mass
1645: //
1646: MTensor<std::complex<double>> MTensorPomeron::iG_PPT_total(const M4Vec &q1, const M4Vec &q2,
1647:                                                            double       M0,
1648:                                                            const std::vector<double> &g_PPT) const {
1649:   // Form FACTOR
1650:   const double LAMBDA  = 1.0;  // GeV
1651:   const double F_tilde = FM(q1.M2(), 1.0) * FM(q2.M2(), 1.0) * F((q1 + q2).M2(), M0, LAMBDA);
1652:
1653:   // Kinematics independent tensor structure [0]
1654:   static const MTensor<std::complex<double>> iG0 = iG_PPT_00();
1655:
1656:   // Kinematics dependent tensor structures  [1 ... 6]
1657:   std::vector<MTensor<std::complex<double>>> iGx;
1658:
1659:   const double EPSILON = 1e-6;
1660:   // For speed, we do not calculate unnecessary here
```



```
1661:   if (std::abs(g_PPT[1]) > EPSILON) iGx.push_back(iG_PPT_12(q1, q2, g_PPT[1], 1));
1662:
1663:   if (std::abs(g_PPT[2]) > EPSILON) iGx.push_back(iG_PPT_12(q1, q2, g_PPT[2], 2));
1664:
1665:   if (std::abs(g_PPT[3]) > EPSILON) iGx.push_back(iG_PPT_03(q1, q2, g_PPT[3]));
1666:
1667:   if (std::abs(g_PPT[4]) > EPSILON) iGx.push_back(iG_PPT_04(q1, q2, g_PPT[4]));
1668:
1669:   if (std::abs(g_PPT[5]) > EPSILON) iGx.push_back(iG_PPT_05(q1, q2, g_PPT[5]));
1670:
1671:   if (std::abs(g_PPT[6]) > EPSILON) iGx.push_back(iG_PPT_06(q1, q2, g_PPT[6]));
1672:
1673:   // Construct total vertex by summing up the tensors
1674:   MTensor<std::complex<double>> T = MTensor({4, 4, 4, 4, 4, 4}, std::complex<double>(0.0));
1675:
1676:   FOR_EACH_6(LI);
1677:
1678:   // Indexing
1679:   const std::vector<size_t> ind = {u, v, k, l, r, s};
1680:
1681:   // For number [0], we apply coupling here
1682:   T(ind) = iG0(ind) * g_PPT[0];
1683:
1684:   // For number [1-6], couplings have been already applied, just loop over
1685:   for (const auto &i : indices(iGx)) { T(ind) += iGx[i](ind); }
1686:
1687:   // Apply form factors
1688:   T(ind) *= F_tilde;
1689:   FOR_EACH_6_END;
1690:
1691:   return T;
1692: }
1693:
1694: // Pomeron-Pomeron-Tensor coupling structure #0
1695: // iG_{\mu \nu \kappa \lambda \rho \sigma} ~ (1,s) = (0,2)
1696: //
1697: // This is kinematics independent, can be pre-calculated.
1698: //
1699: // Remember to apply g_PPT (coupling constant) outside this
1700: //
1701: MTensor<std::complex<double>> MTensorPomeron::iG_PPT_00() const {
1702:   const double               S0     = 1.0;  // Mass scale (GeV)
1703:   const std::complex<double> FACTOR = 2.0 * zi * S0;
1704:
1705:   // Init with zeros!
1706:   MTensor<std::complex<double>> T = MTensor({4, 4, 4, 4, 4, 4}, std::complex<double>(0.0));
1707:
1708:   FOR_EACH_6(LI);
1709:   const std::vector<size_t> ind = {u, v, k, l, r, s};
1710:
1711:   for (auto const &v1 : LI) {
1712:     for (auto const &a1 : LI) {
1713:       for (auto const &l1 : LI) {
1714:         for (auto const &r1 : LI) {
1715:           for (auto const &s1 : LI) {
1716:             for (auto const &u1 : LI) {
1717:               if (v1 != a1 || l1 != r1 || s1 != u1) { continue; }  // speed it up
1718:
1719:               // Note +=
1720:               T(ind) += FACTOR * R_DDDD(u, v, u1, v1) * R_DDDD(k, l, a1, l1) *
1721:                         R_DDDD(r, s, r1, s1) * g[v1][a1] * g[l1][r1] * g[s1][u1];
1722:             }
1723:           }
1724:         }
1725:       }
1726:     }
1727:   }
1728:   FOR_EACH_6_END;
1729:
1730:   return T;
1731: }
1732:
1733: // Pomeron-Pomeron-Tensor coupling structures #1 and #2
1734: // iG_{\mu \nu \kappa \lambda \rho \sigma}
1735: // ~ (1,s) = (2,0) - (2,2)
1736: // ~ (1,s) = (2,0) + (2,2)
1737: //
1738: MTensor<std::complex<double>> MTensorPomeron::iG_PPT_12(const M4Vec &q1, const M4Vec &q2,
1739:                                                         double g_PPT, int mode) const {
1740:   if (!(mode == 1 || mode == 2)) {
1741:     throw std::invalid_argument("MTensorPomeron::iG_PPT_12: Error, mode should be 1 or 2");
1742:   }
1743:   const double               S0     = 1.0;  // Mass scale (GeV)
```



```
1744:   const double               q1q2   = q1 * q2;
1745:   const std::complex<double> FACTOR = -2.0 * zi / S0 * g_PPT;
1746:
1747:   // Coupling structure 1 or 2
1748:   const double sign = (mode == 1) ? -1.0 : 1.0;
1749:
1750:   // Init with zeros!
1751:   MTensor<std::complex<double>> T = MTensor({4, 4, 4, 4, 4, 4}, std::complex<double>(0.0));
1752:   FOR_EACH_6(LI);
1753:   const std::vector<size_t> ind = {u, v, k, l, r, s};
1754:
1755:   // Pre-calculated tensor contraction structures T1,T2,T2 used here
1756:   T(ind) += q1q2 * T1(ind);
1757:
1758:   for (auto const &u1 : LI) {
1759:     for (auto const &r1 : LI) {
1760:       // Note +=
1761:       T(ind) += sign * q2[u1] * (q1 % r1) * T2({u, v, k, l, r, s, u1, r1});
1762:     }
1763:   }
1764:   for (auto const &u1 : LI) {
1765:     for (auto const &s1 : LI) {
1766:       // Note +=
1767:       T(ind) += sign * q1[u1] * (q2 % s1) * T3({u, v, k, l, r, s, u1, s1});
1768:     }
1769:   }
1770:   for (auto const &r1 : LI) {
1771:     for (auto const &s1 : LI) {
1772:       // Note +=
1773:       T(ind) += (q1 % r1) * (q2 % s1) * R_DDDD(u, v, k, l) * R_DDUU(r, s, r1, s1);
1774:     }
1775:   }
1776:
1777:   T(ind) *= FACTOR;
1778:   FOR_EACH_6_END;
1779:
1780:   return T;
1781: }
1782:
1783: // Pomeron-Pomeron-Tensor coupling structure #3
1784: // iG_{\mu \nu \kappa \lambda \rho \sigma} ~ (1,s) = (2,4)
1785: //
1786: MTensor<std::complex<double>> MTensorPomeron::iG_PPT_03(const M4Vec &q1, const M4Vec &q2,
1787:                                                         double g_PPT) const {
1788:   const double               S0     = 1.0;  // Mass scale (GeV)
1789:   const std::complex<double> FACTOR = -zi / S0 * g_PPT;
1790:
1791:   Tensor3<double, 4, 4, 4> A;
1792:   Tensor3<double, 4, 4, 4> B;
1793:   Tensor3<double, 4, 4, 4> C;
1794:   Tensor3<double, 4, 4, 4> D;
1795:
1796:   // Manual contractions
1797:   FOR_EACH_3(LI);
1798:   A(u, v, k) = 0.0;
1799:   B(u, v, k) = 0.0;
1800:   for (auto const &x : LI) {
1801:     // Note +=
1802:     A(u, v, k) += q2[x] * R_DDDD(u, v, x, k);
1803:     B(u, v, k) += q1[x] * R_DDDD(u, v, x, k);
1804:   }
1805:   FOR_EACH_3_END;
1806:
1807:   // Final rank-6 tensor
1808:   MTensor<std::complex<double>> T = MTensor({4, 4, 4, 4, 4, 4}, std::complex<double>(0.0));
1809:   FOR_EACH_6(LI);
1810:   const std::vector<size_t> ind = {u, v, k, l, r, s};
1811:   for (auto const &v1 : LI) {
1812:     for (auto const &l1 : LI) {
1813:       // Note +=
1814:       T(ind) += (A(u, v, v1) * B(k, l, l1) + A(u, v, l1) * B(k, l, v1)) * R_UUDD(v1, l1, r, s);
1815:     }
1816:   }
1817:   T(ind) *= FACTOR;
1818:   /*
1819:
1820:   // NAIVE LOOP
1821:   for (auto const &v1 : LI) {
1822:     for (auto const &a1 : LI) {
1823:       for (auto const &l1 : LI) {
1824:         for (auto const &u1 : LI) {
1825:
1826:           // Note +=
```

```
1827:             T(ind) += FACTOR * (q2[u1] * R_DDDD(u,v,u1,v1) * q1[a1] * R_DDDD(k,l,a1,l1) +
1828:                                 q2[a1] * R_DDDD(u,v,a1,l1) * q1[u1] * R_DDDD(k,l,u1,v1)) *
1829:   R_UUDD(v1,l1,r,s);
1830:       }
1831:     }
1832:   }
1833:   */
1834:   FOR_EACH_6_END;
1835:
1836:   return T;
1837: }
1838:
1839: // Pomeron-Pomeron-Tensor coupling structure #4
1840: // iG_{\mu \nu \kappa \lambda \rho \sigma} ~ (1,s) = (4,2)
1841: //
1842: MTensor<std::complex<double>> MTensorPomeron::iG_PPT_04(const M4Vec &q1, const M4Vec &q2,
1843:                                                         double g_PPT) const {
1844:   const double              S0     = 1.0;  // Mass scale (GeV)
1845:   const std::complex<double> FACTOR = -2.0 * zi / math::pow3(S0) * g_PPT;
1846:   const double              q1q2   = q1 * q2;
1847:
1848:   Tensor1<double, 4> q1_D = {q1 % 0, q1 % 1, q1 % 2, q1 % 3};
1849:   Tensor1<double, 4> q2_D = {q2 % 0, q2 % 1, q2 % 2, q2 % 3};
1850:
1851:   Tensor4<double, 4, 4, 4, 4> A;
1852:   Tensor4<double, 4, 4, 4, 4> B;
1853:   Tensor2<double, 4, 4>       C;
1854:
1855:   // Manual contractions (not possible with autocontraction)
1856:   FOR_EACH_4(LI);
1857:   A(u, v, k, l) = 0.0;
1858:   B(u, v, k, l) = 0.0;
1859:
1860:   for (auto const &u1 : LI) {
1861:     for (auto const &v1 : LI) {
1862:       for (auto const &a1 : LI) {
1863:         // Note +=
1864:         A(u, v, k, l) += q2[v1] * R_DDDD(u, v, v1, a1) * q1[u1] * R_DDDD(k, l, u1, a1);
1865:         B(u, v, k, l) += q2[u1] * R_DDDD(u, v, u1, a1) * q1[v1] * R_DDDD(k, l, v1, a1);
1866:       }
1867:     }
1868:   }
1869:   FOR_EACH_4_END;
1870:
1871:   // Autocontractions
1872:   {
1873:     FTensor::Index<'a', 4> r;
1874:     FTensor::Index<'b', 4> s;
1875:     FTensor::Index<'c', 4> a1;
1876:     FTensor::Index<'d', 4> l1;
1877:
1878:     C(r, s) = q1_D(a1) * q2_D(l1) * R_UUDD(a1, l1, r, s);
1879:   }
1880:
1881:   // Final rank-6 expression
1882:   MTensor<std::complex<double>> T = MTensor({4, 4, 4, 4, 4, 4}, std::complex<double>(0.0));
1883:   FOR_EACH_6(LI);
1884:   T((u, v, k, l, r, s)) =
1885:       FACTOR * (A(u, v, k, l) + B(u, v, k, l) - 2.0 * q1q2 * R_DDDD(u, v, k, l)) * C(r, s);
1886:
1887:   /*
1888:   // NAIVE LOOP
1889:   for (auto const &v1 : LI) {
1890:     for (auto const &a1 : LI) {
1891:       for (auto const &l1 : LI) {
1892:         for (auto const &u1 : LI) {
1893:
1894:           // Note +=
1895:           T(ind) += FACTOR * (q2[v1] * R_DDDD(u,v,v1,a1) * q1[u1] * R_DDDD(k,l,u1,a1)
1896:                               + q2[u1] * R_DDDD(u,v,u1,a1) * q1[v1] * R_DDDD(k,l,v1,a1)
1897:                               - 2.0 * q1q2 * R_DDDD(u,v,k,l)) * (q1 % a1) * (q2 % l1) *
1898:   R_UUDD(a1,l1,r,s);
1899:         }
1900:       }
1901:     }
1902:   }
1903:   */
1904:   FOR_EACH_6_END;
1905:   return T;
1906: }
1907:
1908:
1909: // Pomeron-Pomeron-Tensor coupling structure #05
```



```
1910: // iG_{\mu \nu \kappa \lambda \rho \sigma} ~ (1,s) = (4,4)
1911: //
1912: MTensor<std::complex<double>> MTensorPomeron::iG_PPT_05(const M4Vec &q1, const M4Vec &q2,
1913:                                                         double g_PPT) const {
1914:   const double              S0     = 1.0;  // Mass scale (GeV)
1915:   const std::complex<double> FACTOR = zi / math::pow3(S0) * g_PPT;
1916:
1917:   Tensor1<double, 4> q1_U = {q1[0], q1[1], q1[2], q1[3]};
1918:   Tensor1<double, 4> q2_U = {q2[0], q2[1], q2[2], q2[3]};
1919:   Tensor1<double, 4> q1_D = {q1 % 0, q1 % 1, q1 % 2, q1 % 3};
1920:   Tensor1<double, 4> q2_D = {q2 % 0, q2 % 1, q2 % 2, q2 % 3};
1921:
1922:   Tensor4<double, 4, 4, 4, 4> A;
1923:   Tensor2<double, 4, 4>       B;
1924:   Tensor4<double, 4, 4, 4, 4> C;
1925:   Tensor2<double, 4, 4>       D;
1926:
1927:   // Manual contractions (not possible with autocontraction)
1928:   MTensor<std::complex<double>> T = MTensor({4, 4, 4, 4, 4, 4}, std::complex<double>(0.0));
1929:   FOR_EACH_6(LI);
1930:   A(u, v, r, s) = 0.0;
1931:   C(k, l, r, s) = 0.0;
1932:
1933:   for (auto const &u1 : LI) {
1934:     for (auto const &v1 : LI) {
1935:       for (auto const &r1 : LI) {
1936:         // Note +=
1937:         A(u, v, r, s) += q2_U(u1) * R_DDDD(u, v, u1, v1) * q2_D(r1) * R_UUDD(v1, r1, r, s);
1938:         C(k, l, r, s) += q1_U(u1) * R_DDDD(k, l, u1, v1) * q1_D(r1) * R_UUDD(v1, r1, r, s);
1939:       }
1940:     }
1941:   }
1942:   FOR_EACH_6_END;
1943:
1944:   // Autocontractions
1945:   {
1946:     FTensor::Index<'a', 4> u;
1947:     FTensor::Index<'b', 4> v;
1948:     FTensor::Index<'c', 4> k;
1949:     FTensor::Index<'d', 4> l;
1950:     FTensor::Index<'j', 4> a1;
1951:     FTensor::Index<'k', 4> l1;
1952:
1953:     B(k, l) = q1_U(a1) * q1_U(l1) * R_DDDD(k, l, a1, l1);
1954:     D(u, v) = q2_U(a1) * q2_U(l1) * R_DDDD(u, v, a1, l1);
1955:   }
1956:
1957:   // Final rank-6 expression
1958:   FOR_EACH_6(LI);
1959:   T((u, v, k, l, r, s)) = FACTOR * (A(u, v, r, s) * B(k, l) + C(k, l, r, s) * D(u, v));
1960:
1961:   /*
1962:   // NAIVE LOOP
1963:   for (auto const &v1 : LI) {
1964:     for (auto const &a1 : LI) {
1965:       for (auto const &l1 : LI) {
1966:         for (auto const &u1 : LI) {
1967:           for (auto const &r1 : LI) {
1968:
1969:             // Note +=
1970:             T(ind) += FACTOR * (q2[u1] * (q2 % r1) * R_DDDD(u,v,u1,v1) * R_UUDD(v1,r1,r,s) * q1[a1]
1971:   * q1[l1] * R_DDDD(k,l,a1,l1)
1972:                               + q1[u1] * (q1 % r1) * R_DDDD(k,l,u1,v1) * R_UUDD(v1,r1,r,s) * q2[a1]
1973:   * q2[l1] * R_DDDD(u,v,a1,l1));
1974:           }
1975:         }
1976:       }
1977:     }
1978:   }
1979:   */
1980:   FOR_EACH_6_END;
1981:   return T;
1982: }
1983:
1984: // Pomeron-Pomeron-Tensor coupling structure #6
1985: // iG_{\mu \nu \kappa \lambda \rho \sigma} ~ (1,s) = (6,4)
1986: //
1987: MTensor<std::complex<double>> MTensorPomeron::iG_PPT_06(const M4Vec &q1, const M4Vec &q2,
1988:                                                         double g_PPT) const {
1989:   const double              S0     = 1.0;  // Mass scale (GeV)
1990:   const std::complex<double> FACTOR = -2.0 * zi / math::pow5(S0) * g_PPT;
1991:
1992:   FTensor::Index<'a', 4> a;
```



```
1993:   FTensor::Index<'b', 4> b;
1994:   FTensor::Index<'c', 4> c;
1995:   FTensor::Index<'d', 4> d;
1996:
1997:   Tensor1<double, 4> q1_U = {q1[0], q1[1], q1[2], q1[3]};
1998:   Tensor1<double, 4> q2_U = {q2[0], q2[1], q2[2], q2[3]};
1999:
2000:   Tensor2<double, 4, 4> A;
2001:   Tensor2<double, 4, 4> B;
2002:   Tensor2<double, 4, 4> C;
2003:
2004:   // Autocontractions
2005:   A(a, b) = q2_U(c) * q2_U(d) * R_DDDD(a, b, c, d);
2006:   B(a, b) = q1_U(c) * q1_U(d) * R_DDDD(a, b, c, d);
2007:   C(a, b) = q1_U(c) * q2_U(d) * R_DDDD(a, b, c, d);
2008:
2009:   MTensor<std::complex<double>> T = MTensor({4, 4, 4, 4, 4, 4}, std::complex<double>(0.0));
2010:   FOR_EACH_6(LI);
2011:   T((u, v, k, l, r, s)) = FACTOR * A(u, v) * B(k, l) * C(r, s);
2012:   FOR_EACH_6_END;
2013:
2014:   return T;
2015: }
2016:
2017:
2018: // ================================================================
2019:
2020:
2021: // f0 (Scalar) - (Pseudo)Scalar - (Pseudo)Scalar vertex function
2022: // i\Gamma
2023: //
2024: // Input as contravariant (upper index) 4-vectors, meson on-shell mass M0
2025: //
2026: // Relations: p3_\mu iG^{\mu \nu} = 0, p4_\nu iG^{\mu \nu} = 0
2027: //
2028: std::complex<double> MTensorPomeron::iG_f0ss(const M4Vec &p3, const M4Vec &p4, double M0,
2029:                                              double g1) const {
2030:   const double S0 = 1.0;  // Mass scale (GeV)
2031:
2032:   // Form FACTOR
2033:   const double lambda4 = 1.0;  // GeV^4 (normalization scale)
2034:   auto         F_f0ss  = [&](double q2) { return std::exp(-pow2(q2 - M0 * M0) / lambda4); };
2035:
2036:   return zi * g1 * S0 * F_f0ss((p3 + p4).M2());
2037: }
2038:
2039: // f0 (Scalar) - Vector (Massive) - Vector (Massive) vertex function
2040: // i\Gamma_{\mu \nu}
2041: //
2042: // Input as contravariant (upper index) 4-vectors, meson on-shell mass M0
2043: //
2044: // Relations: p3_\mu iG^{\mu \nu} = 0, p4_\nu iG^{\mu \nu} = 0
2045: //
2046: Tensor2<std::complex<double>, 4, 4> MTensorPomeron::iG_f0vv(const M4Vec &p3, const M4Vec &p4,
2047:                                                             double M0, double g1, double g2) const {
2048:   const double S0     = 1.0;  // Mass scale (GeV)
2049:   const double LAMBDA = 1.0;  // GeV
2050:
2051:   const double p3p4 = p3 * p4;
2052:   const double p3sq = p3.M2();
2053:   const double p4sq = p4.M2();
2054:
2055:   const double               F_ = FM(p3sq, 0.5) * FM(p4sq, 0.5) * F((p3 + p4).M2(), M0, LAMBDA);
2056:   const std::complex<double> FACTOR1 = zi * g1 * 2.0 / math::pow3(S0) * F_;
2057:   const std::complex<double> FACTOR2 = zi * g2 * 2.0 / S0 * F_;
2058:
2059:   Tensor2<std::complex<double>, 4, 4> T;
2060:   FOR_EACH_2(LI);
2061:   T(u, v) = FACTOR1 * (p3sq * p4sq * g[u][v] - p4sq * (p3 % u) * (p3 % v) -
2062:                        p3sq * (p4 % u) * (p4 % v) + p3p4 * (p3 % u) * (p4 % v)) +
2063:             FACTOR2 * ((p4 % u) * (p3 % v) - p3p4 * g[u][v]);
2064:   FOR_EACH_2_END;
2065:
2066:   return T;
2067: }
2068:
2069: // Vector (Massive) - Pseudoscalar - Pseudoscalar vertex function
2070: // i\Gamma_\mu(k1,k2)
2071: //
2072: // Input as contravariant (upper index) 4-vectors
2073: //
2074: Tensor1<std::complex<double>, 4> MTensorPomeron::iG_vpsps(const M4Vec &k1, const M4Vec &k2,
2075:                                                           double M0, double g1) const {
```



```
2076:   const M4Vec p = k1 - k2;
2077:
2078:   // Form FACTOR
2079:   const double LAMBDA = 1.0;  // GeV
2080:   const double F_     = FM(k1.M2(), 0.5) * FM(k2.M2(), 0.5) * F((k1 + k2).M2(), M0, LAMBDA);
2081:   const std::complex<double> FACTOR = -0.5 * zi * g1 * F_;
2082:
2083:   Tensor1<std::complex<double>, 4> T;
2084:   for (const auto &mu : LI) { T(mu) = FACTOR * (p % mu); }
2085:   return T;
2086: }
2087:
2088: // pseudoscalar - vector - vector vertex function
2089: // i\Gamma_{\kappa\lambda}(k1,k2)
2090: //
2091: // Input as contravariant (upper index) 4-vectors, M0 is the pseudoscalar on-shell mass
2092: //
2093: Tensor2<std::complex<double>, 4, 4> MTensorPomeron::iG_psvv(const M4Vec &p3, const M4Vec &p4,
2094:                                                             double M0, double g1) const {
2095:   // Form FACTOR
2096:   const double LAMBDA = 1.0;  // GeV
2097:
2098:   const double               S0     = 1.0;  // Mass scale (GeV)
2099:   const std::complex<double> FACTOR = zi * g1 / (2 * S0) * F((p3 + p4).M2(), M0, LAMBDA);
2100:
2101:   Tensor1<std::complex<double>, 4> p3_ = {p3[0], p3[1], p3[2], p3[3]};
2102:   Tensor1<std::complex<double>, 4> p4_ = {p4[0], p4[1], p4[2], p4[3]};
2103:
2104:   FTensor::Index<'a', 4> mu;
2105:   FTensor::Index<'b', 4> nu;
2106:   FTensor::Index<'c', 4> rho;
2107:   FTensor::Index<'d', 4> sigma;
2108:
2109:   // Contract
2110:   Tensor2<std::complex<double>, 4, 4> T;
2111:   T(mu, nu) = FACTOR * p3_(rho) * p4_(sigma) * eps_lo(mu, nu, rho, sigma);
2112:
2113:   return T;
2114: }
2115:
2116: // f2 - pseudoscalar - pseudoscalar vertex function
2117: // i\Gamma_{\kappa\lambda}(k1,k2)
2118: //
2119: // Input as contravariant (upper index) 4-vectors, M0 is the f2 meson on-shell mass
2120: //
2121: Tensor2<std::complex<double>, 4, 4> MTensorPomeron::iG_f2psps(const M4Vec &k1, const M4Vec &k2,
2122:                                                               double M0, double g1) const {
2123:   // Form FACTOR
2124:   const double              LAMBDA = 1.0;  // GeV
2125:   const double              S0     = 1.0;  // Mass scale (GeV)
2126:   const M4Vec               k1_k2  = k1 - k2;
2127:   const std::complex<double> FACTOR = -zi * g1 / (2 * S0) * F((k1 + k2).M2(), M0, LAMBDA);
2128:
2129:   const double k1_k2_sq = k1_k2.M2();
2130:
2131:   Tensor2<std::complex<double>, 4, 4> T;
2132:   FOR_EACH_2(LI);
2133:   T(u, v) = FACTOR * ((k1_k2 % u) * (k1_k2 % v) - 0.25 * g[u][v] * k1_k2_sq);
2134:   FOR_EACH_2_END;
2135:
2136:   return T;
2137: }
2138:
2139: // f2 - Vector (massive) - Vector (Massive) vertex function
2140: // i\Gamma_{\mu\nu\kappa\lambda}(k1,k2)
2141: //
2142: // Input as contravariant (upper index) 4-vectors, M0 is the f2 meson on-shell mass
2143: //
2144: Tensor4<std::complex<double>, 4, 4, 4, 4> MTensorPomeron::iG_f2vv(const M4Vec &k1, const M4Vec &k2,
2145:                                                                   double M0, double g1,
2146:                                                                   double g2) const {
2147:   const double S0     = 1.0;  // Mass scale (GeV)
2148:   const double LAMBDA = 1.0;  // GeV
2149:
2150:   const Tensor4<std::complex<double>, 4, 4, 4, 4> G0 = Gamma0(k1, k2);
2151:   const Tensor4<std::complex<double>, 4, 4, 4, 4> G2 = Gamma2(k1, k2);
2152:
2153:   const std::complex<double> F_ =
2154:       FM(k1.M2(), 0.5) * FM(k2.M2(), 0.5) * F((k1 + k2).M2(), M0, LAMBDA);
2155:   const std::complex<double> FACTOR1 = zi * 2.0 / math::pow3(S0) * g1 * F_;
2156:   const std::complex<double> FACTOR2 = -zi * 1.0 / S0 * g2 * F_;
2157:
2158:   Tensor4<std::complex<double>, 4, 4, 4, 4> T;
```

```
2159:   FOR_EACH_4(LI);
2160:   T(u, v, k, l) = FACTOR1 * G0(u, v, k, l) + FACTOR2 * G2(u, v, k, l);
2161:   FOR_EACH_4_END;
2162:
2163:   return T;
2164: }
2165:
2166: // f2 - gamma - gamma vertex function
2167: // i\Gamma_{\mu\nu\kappa\lambda}(k1,k2)
2168: //
2169: // Input as contravariant (upper index) 4-vectors, M0 is the f2 meson on-shell mass
2170: // Couplings g1,g2
2171: //
2172: // Example couplings:
2173: //
2174: // const double e = msqrt(qed::alpha_QED() * 4.0 * PI);      // ~ 0.3, no running
2175: // const double a_f2yy = pow2(e) / (4 * PI) * 1.45;  // GeV^(-3)
2176: // const double b_f2yy = pow2(e) / (4 * PI) * 2.49;  // GeV^(-1)
2177:
2178: Tensor4<std::complex<double>, 4, 4, 4, 4> MTensorPomeron::iG_f2yy(const M4Vec &k1, const M4Vec &k2,
2179:                                                                   double M0, double g1,
2180:                                                                   double g2) const {
2181:   // Form FACTOR
2182:   const double LAMBDA = 1.0;  // GeV
2183:
2184:   const Tensor4<std::complex<double>, 4, 4, 4, 4> G0 = Gamma0(k1, k2);
2185:   const Tensor4<std::complex<double>, 4, 4, 4, 4> G2 = Gamma2(k1, k2);
2186:
2187:   const std::complex<double> FACTOR =
2188:       zi * FM(k1.M2(), 0.5) * FM(k2.M2(), 0.5) * F((k1 + k2).M2(), M0, LAMBDA);
2189:
2190:   Tensor4<std::complex<double>, 4, 4, 4, 4> T;
2191:   FOR_EACH_4(LI);
2192:   T(u, v, k, l) = FACTOR * (2.0 * g1 * G0(u, v, k, l) - g2 * G2(u, v, k, l));
2193:   FOR_EACH_4_END;
2194:
2195:   return T;
2196: }
2197:
2198: // Pomeron-Pseudoscalar-Pseudoscalar vertex function
2199: // i\Gamma_{\mu\nu}(p',p)
2200: //
2201: // Input as contravariant (upper index) 4-vectors
2202: //
2203: Tensor2<std::complex<double>, 4, 4> MTensorPomeron::iG_Ppsps(const M4Vec &prime, const M4Vec &p,
2204:                                                              double g1) const {
2205:   const M4Vec                psum   = prime + p;
2206:   const double               M2     = psum.M2();
2207:   const std::complex<double> FACTOR = -zi * 2.0 * g1 * FM((prime - p).M2(), 0.5);
2208:
2209:   Tensor2<std::complex<double>, 4, 4> T;
2210:   FOR_EACH_2(LI);
2211:   T(u, v) = FACTOR * ((psum % u) * (psum % v) - 0.25 * g[u][v] * M2);
2212:   FOR_EACH_2_END;
2213:
2214:   return T;
2215: }
2216:
2217: // Pomeron-Vector(massive)-Vector(massive)  vertex function
2218: // i\Gamma_{\alpha\beta\gamma\delta}(p',p)
2219: //
2220: // Input M0 is the vector meson on-shell mass
2221: //
2222: // Input as contravariant (upper index) 4-vectors
2223: //
2224: Tensor4<std::complex<double>, 4, 4, 4, 4> MTensorPomeron::iG_Pvv(const M4Vec &prime, const M4Vec &p,
2225:                                                                   double g1, double g2) const {
2226:   const Tensor4<std::complex<double>, 4, 4, 4, 4> G0 = Gamma0(prime, -p);
2227:   const Tensor4<std::complex<double>, 4, 4, 4, 4> G2 = Gamma2(prime, -p);
2228:
2229:   const std::complex<double> FACTOR = zi * FM((prime - p).M2(), 0.5);
2230:
2231:   Tensor4<std::complex<double>, 4, 4, 4, 4> T;
2232:   FOR_EACH_4(LI);
2233:   T(u, v, k, l) = FACTOR * (2.0 * g1 * G0(u, v, k, l) - g2 * G2(u, v, k, l));
2234:   FOR_EACH_4_END;
2235:
2236:   return T;
2237: }
2238:
2239:
2240: // Gamma-Vector meson transition vertex
2241: // iGamma_{\mu \nu}
```



```
2242: //
2243: // pdg = 113 / "rho0", 223 / "omega0", 333 / "phi0"
2244: //
2245: Tensor2<std::complex<double>, 4, 4> MTensorPomeron::iG_yV(double q2, int pdg) const {
2246:   const double e      = msqrt(qed::alpha_QED() * 4.0 * PI);  // ~ 0.3, no running
2247:   double       gammaV = 0.0;
2248:   double       mV     = 0.0;
2249:
2250:   // Vector Meson Dominance (VMD) type parameters
2251:   if (pdg == 113) {  // rho
2252:     gammaV = msqrt(4 * PI / 0.496);
2253:     mV     = 0.770;
2254:   } else if (pdg == 223) {  // omega
2255:     gammaV = msqrt(4 * PI / 0.042);
2256:     mV     = 0.785;
2257:   } else if (pdg == 333) {  // phi
2258:     gammaV = (-1.0) * msqrt(4 * PI / 0.0716);
2259:     mV     = 1.020;
2260:   } else {
2261:     throw std::invalid_argument("MTensorPomeron::iG_yV: Unknown vector meson pdg = " +
2262:                                 std::to_string(pdg));
2263:   }
2264:
2265:   Tensor2<std::complex<double>, 4, 4> T;
2266:   FOR_EACH_2(LI);
2267:   T(u, v) = -zi * e * pow2(mV) / gammaV * g[u][v];
2268:   FOR_EACH_2_END;
2269:
2270:   return T;
2271: }
2272:
2273: // ---------------------------------------------------------------------
2274: // Propagators
2275:
2276: // Tensor Pomeron propagator: (covariant == contravariant)
2277: //
2278: // i\Delta_{\mu\nu,\kappa\lambda}(s,t) = i\Delta^{\mu\nu,\kappa\lambda}(s,t)
2279: //
2280: // Input as (sub)-Mandelstam invariants: s,t
2281: //
2282: // Symmetry relations:
2283: // \Delta_{\mu\nu,\kappa\lambda} = \Delta_{\nu\mu,\kappa\lambda} =
2284: // \Delta_{\mu\nu,\lambda\kappa} = \Delta_{\kappa\lambda,\mu\nu}
2285: //
2286: // Contraction with g^{\mu\nu} or g^{\kappa\lambda} gives 0
2287: //
2288: Tensor4<std::complex<double>, 4, 4, 4, 4> MTensorPomeron::iD_P(double s, double t) const {
2289:   const std::complex<double> FACTOR =
2290:       1.0 / (4.0 * s) * std::pow(-zi * s * Param.ap_P, alpha_P(t) - 1.0);
2291:
2292:   Tensor4<std::complex<double>, 4, 4, 4, 4> T;
2293:   FOR_EACH_4(LI);
2294:   T(u, v, k, l) = FACTOR * (g[u][k] * g[v][l] + g[u][l] * g[v][k] - 0.5 * g[u][v] * g[k][l]);
2295:   FOR_EACH_4_END;
2296:
2297:   return T;
2298: }
2299:
2300: // Tensor Reggeon propagator iD (f2-a2-reggeon)
2301: //
2302: // Input as (sub)-Mandelstam invariants: s,t
2303: //
2304: Tensor4<std::complex<double>, 4, 4, 4, 4> MTensorPomeron::iD_2R(double s, double t) const {
2305:   const std::complex<double> FACTOR =
2306:       1.0 / (4.0 * s) * std::pow(-zi * s * Param.ap_2R, alpha_2R(t) - 1.0);
2307:
2308:   Tensor4<std::complex<double>, 4, 4, 4, 4> T;
2309:   FOR_EACH_4(LI);
2310:   T(u, v, k, l) = FACTOR * (g[u][k] * g[v][l] + g[u][l] * g[v][k] - 0.5 * g[u][v] * g[k][l]);
2311:   FOR_EACH_4_END;
2312:
2313:   return T;
2314: }
2315:
2316: // Vector Reggeon propagator iD (rho-reggeon, omega-reggeon)
2317: //
2318: // Input as (sub)-Mandelstam invariants s,t
2319: //
2320: Tensor2<std::complex<double>, 4, 4> MTensorPomeron::iD_1R(double s, double t) const {
2321:   const std::complex<double> FACTOR =
2322:       zi / pow2(Param.M_1R) * std::pow(-zi * s * Param.ap_1R, alpha_1R(t) - 1.0);
2323:
2324:   Tensor2<std::complex<double>, 4, 4> T;
```



```
2325:   FOR_EACH_2(LI);
2326:   T(u, v) = FACTOR * g[u][v];
2327:   FOR_EACH_2_END;
2328:
2329:   return T;
2330: }
2331:
2332: // Odderon propagator iD
2333: //
2334: // Input as (sub)-Mandelstam invariants s,t
2335: //
2336: Tensor2<std::complex<double>, 4, 4> MTensorPomeron::iD_O(double s, double t) const {
2337:   const std::complex<double> FACTOR =
2338:       -zi * Param.eta_O / pow2(Param.M_O) * std::pow(-zi * s * Param.ap_O, alpha_O(t) - 1.0);
2339:
2340:   Tensor2<std::complex<double>, 4, 4> T;
2341:   FOR_EACH_2(LI);
2342:   T(u, v) = FACTOR * g[u][v];
2343:   FOR_EACH_2_END;
2344:
2345:   return T;
2346: }
2347:
2348:
2349: // Vector propagator iD_{\mu \nu} == iD^{\mu \nu} with (naive) Reggeization
2350: //
2351: // Input as contravariant 4-vector and the particle peak mass M0
2352: //
2353: Tensor2<std::complex<double>, 4, 4> MTensorPomeron::iD_V(const M4Vec &p, double M0,
2354:                                                          double s34) const {
2355:   const double m2       = p.M2();
2356:   const double alpha    = 0.5 + 0.9 * m2;  // Typical Reggeon trajectory
2357:   const double s_thresh = 4 * pow2(M0);
2358:   const double phi      = PI / 2.0 * std::exp((s_thresh - s34) / s_thresh) - PI / 2.0;
2359:
2360:   // Reggeization
2361:   std::complex<double> REGGEIZE = std::pow(std::exp(zi * phi) * s34 / s_thresh, alpha - 1.0);
2362:
2363:   Tensor2<std::complex<double>, 4, 4> T;
2364:   FOR_EACH_2(LI);
2365:   T(u, v) = -g[u][v] / (m2 - pow2(M0)) * REGGEIZE;
2366:   FOR_EACH_2_END;
2367:
2368:   return T;
2369: }
2370:
2371: // Scalar propagator iD with zero width
2372: //
2373: // Input as 4-vector and the particle peak mass M0
2374: //
2375: std::complex<double> MTensorPomeron::iD_MES(const M4Vec &p, double M0) const {
2376:   return zi / (p.M2() - pow2(M0));
2377: }
2378:
2379: // Scalar propagator iD with finite Breit-Wigner width
2380: //
2381: // Input as 4-vector and the particle peak mass M0 and full width
2382: //
2383: std::complex<double> MTensorPomeron::iD_MES(const M4Vec &p, double M0, double Gamma) const {
2384:   const std::complex<double> Delta = 1.0 / (p.M2() - pow2(M0) + zi * M0 * Gamma);
2385:
2386:   return zi * Delta;
2387: }
2388:
2389: // Massive vector propagator iD_{\mu \nu} or iD^{\mu \nu} with INDEX_UP == true
2390: //
2391: // Input as contravariant 4-vector and the particle peak mass M0 and full width Gamma
2392: //
2393: Tensor2<std::complex<double>, 4, 4> MTensorPomeron::iD_VMES(const M4Vec &p, double M0, double Gamma,
2394:                                                             bool INDEX_UP) const {
2395:   // Transverse part
2396:   const double               m2     = p.M2();
2397:   const std::complex<double> Delta_T = 1.0 / (m2 - pow2(M0) + zi * M0 * Gamma);
2398:
2399:   // Longitudinal part (does not enter here)
2400:   const double Delta_L = 0.0;
2401:
2402:   Tensor2<std::complex<double>, 4, 4> T;
2403:   if (INDEX_UP) {
2404:     FOR_EACH_2(LI);
2405:     T(u, v) =
2406:         zi * (-g[u][v] + (p[u]) * (p[v]) / m2) * Delta_T - zi * (p[u]) * (p[v]) / m2 * Delta_L;
2407:     FOR_EACH_2_END;
```



```
2408:   } else {
2409:     FOR_EACH_2(LI);
2410:     T(u, v) =
2411:         zi * (-g[u][v] + (p % u) * (p % v) / m2) * Delta_T - zi * (p % u) * (p % v) / m2 * Delta_L;
2412:     FOR_EACH_2_END;
2413:   }
2414:   return T;
2415: }
2416:
2417: // Massive tensor propagator iD_{\mu \nu \kappa \lambda} or iD^{} with INDEX_UP == true
2418: //
2419: // Input as contravariant (upper index) 4-vector and the particle peak mass M0
2420: // and full width Gamma
2421: //
2422: Tensor4<std::complex<double>, 4, 4, 4, 4> MTensorPomeron::iD_TMES(const M4Vec &p, double M0,
2423:                                                                   double Gamma,
2424:                                                                   bool   INDEX_UP) const {
2425:   const double m2 = p.M2();
2426:
2427:   // Inline auxialary function
2428:   auto ghat = [&](int u, int v) {
2429:     if (INDEX_UP) {
2430:       return -g[u][v] + (p[u]) * (p[v]) / m2;
2431:     } else {
2432:       return -g[u][v] + (p % u) * (p % v) / m2;
2433:     }
2434:   };
2435:
2436:   const std::complex<double> Delta = 1.0 / (m2 - pow2(M0) + zi * M0 * Gamma);
2437:
2438:   Tensor4<std::complex<double>, 4, 4, 4, 4> T;
2439:   FOR_EACH_4(LI);
2440:   T(u, v, k, l) = zi * Delta *
2441:                   (0.5 * (ghat(u, k) * ghat(v, l) + ghat(u, l) * ghat(v, k)) -
2442:                    1.0 / 3.0 * ghat(u, v) * ghat(k, l));
2443:   FOR_EACH_4_END;
2444:
2445:   return T;
2446: }
2447:
2448: // ---------------------------------------------------------------------
2449: // Trajectories (simple linear/affine here)
2450:
2451: // Pomeron trajectory
2452: double MTensorPomeron::alpha_P(double t) const { return 1.0 + Param.delta_P + Param.ap_P * t; }
2453: // Odderon trajectory
2454: double MTensorPomeron::alpha_O(double t) const { return 1.0 + Param.delta_O + Param.ap_O * t; }
2455: // Reggeon (rho,omega) trajectory
2456: double MTensorPomeron::alpha_1R(double t) const { return 1.0 + Param.delta_1R + Param.ap_1R * t; }
2457: // Reggeon (f2,a2) trajectory
2458: double MTensorPomeron::alpha_2R(double t) const { return 1.0 + Param.delta_2R + Param.ap_2R * t; }
2459:
2460: // ---------------------------------------------------------------------
2461: // Form factors
2462:
2463: // Proton electromagnetic Dirac form FACTOR
2464: double MTensorPomeron::F1(double t) const { return form::F1(t); }
2465:
2466: // Proton electromagnetic Pauli form FACTOR
2467: double MTensorPomeron::F2(double t) const { return form::F2(t); }
2468:
2469: // Proton electromagnetic Dirac form FACTOR
2470: double MTensorPomeron::F1_(double t) const {
2471:   return (1 - (t / (4 * pow2(PDG::mp))) * form::mu_ratio()) / (1 - t / (4 * pow2(PDG::mp))) * GD(t);
2472: }
2473:
2474: // Proton electromagnetic Pauli form FACTOR
2475: double MTensorPomeron::F2_(double t) const {
2476:   return (form::mu_ratio() - 1) / (1 - t / (4 * pow2(PDG::mp))) * GD(t);
2477: }
2478:
2479: // Dipole form FACTOR
2480: double MTensorPomeron::GD(double t) const {
2481:   const double m2D = 0.71;  // GeV^2
2482:   return 1.0 / pow2(1 - t / m2D);
2483: }
2484:
2485: // Meson form FACTOR (pion electromagnetic type)
2486: double MTensorPomeron::FM(double q2, double LAMBDA2) const { return 1.0 / (1.0 - q2 / LAMBDA2); }
2487:
2488: // Resonance form factor
2489: double MTensorPomeron::F(double q2, double M0, double LAMBDA) const {
2490:   return std::exp(-pow2(q2 - pow2(M0)) / math::pow4(LAMBDA));
```

./src/MTensorPomeron.cc 31/32

```
}

// ------------------------------------------------------------------------
// Tensor functions

// Output: \Gamma^{(0)}_{\mu\nu\kappa\lambda}
//
// Input must be contravariant (upper index) 4-vectors
//
Tensor4<std::complex<double>, 4, 4, 4, 4> MTensorPomeron::Gamma0(const M4Vec &k1,
                                                                 const M4Vec &k2) const {
  const double                                k1k2 = k1 * k2;  // 4-dot product
  Tensor4<std::complex<double>, 4, 4, 4, 4> T;
  FOR_EACH_4(LI);
  T(u, v, k, l) = (k1k2 * g[u][v] - (k2 % u) * (k1 % v)) *
                  ((k1 % k) * (k2 % l) + (k2 % k) * (k1 % l) - 0.5 * k1k2 * g[k][l]);
  FOR_EACH_4_END;

  return T;
}

// Output: \Gamma^{(2)}_{\mu\nu\kappa\lambda}
//
// Input must be contravariant (upper index) 4-vectors
//
Tensor4<std::complex<double>, 4, 4, 4, 4> MTensorPomeron::Gamma2(const M4Vec &k1,
                                                                 const M4Vec &k2) const {
  const double                                k1k2 = k1 * k2;  // 4-dot product
  Tensor4<std::complex<double>, 4, 4, 4, 4> T;
  FOR_EACH_4(LI);
  T(u, v, k, l) = k1k2 * (g[u][k] * g[v][l] + g[u][l] * g[v][k]) +
                  g[u][v] * ((k1 % k) * (k2 % l) + (k2 % k) * (k1 % l)) -
                  (k1 % v) * (k2 % l) * g[u][k] - (k1 % v) * (k2 % k) * g[u][l] -
                  (k2 % u) * (k1 % l) * g[v][k] - (k2 % u) * (k1 % k) * g[v][l] -
                  (k1k2 * g[u][v] - (k2 % u) * (k1 % v)) * g[k][l];
  FOR_EACH_4_END;

  return T;
}

// Pre-calculate tensors for speed
//
// 1. Minkowski metric tensor
// 2. Auxiliary (helper) tensor
// 1/2 g_{\mu\kappa} g_{\nu\lambda) + 1/2 g_{\mu\lambda}g_{\nu\kappa} - 1/4
// g_{\mu\nu}g_{kappa\lambda}
//
void MTensorPomeron::CalcRTensor() {
  // Minkowski metric tensor (+1,-1,-1,-1)
  FOR_EACH_2(LI);
  gT(u, v) = g[u][v];
  FOR_EACH_2_END;

  // ------------------------------------------------------------------------
  // Covariant [lower index] 4D-epsilon tensor

  const MTensor<int> etensor = math::EpsTensor(4);

  FOR_EACH_4(LI);
  std::vector<std::size_t> ind = {u, v, k, l};
  eps_lo(u, v, k, l)           = static_cast<double>(etensor(ind));
  FOR_EACH_4_END;

  // Free Lorentz indices [second parameter denotes the range of index]
  FTensor::Index<'a', 4> a;
  FTensor::Index<'b', 4> b;
  FTensor::Index<'c', 4> c;
  FTensor::Index<'d', 4> d;
  FTensor::Index<'g', 4> alfa;
  FTensor::Index<'h', 4> beta;

  // Contravariant version (make it in two steps)
  eps_hi(a, b, c, d) = eps_lo(alfa, beta, c, d) * gT(a, alfa) * gT(b, beta);
  eps_hi(a, b, c, d) = eps_hi(a, b, alfa, beta) * gT(c, alfa) * gT(d, beta);

  // ------------------------------------------------------------------------

  // Aux tensor R
  FTensor::Tensor4<double, 4, 4, 4, 4> R;

  R(a, b, c, d) =
      0.5 * gT(a, c) * gT(b, d) + 0.5 * gT(a, d) * gT(b, c) + 0.25 * gT(a, b) * gT(c, d);

```

./src/MTensorPomeron.cc 32/32

```
  // Different mixed index covariant/contravariant versions
  // by contraction with g_{\mu\nu}
  R_DDDD       = R;
  R_DDDU(a, b, c, d) = R(a, b, c, alfa) * gT(d, alfa);
  R_DDUU(a, b, c, d) = R(a, b, alfa, beta) * gT(c, alfa) * gT(d, beta);
  R_UUDD(a, b, c, d) = R(alfa, beta, c, d) * gT(a, alfa) * gT(b, beta);

  // ------------------------------------------------------------------------
  // Pre-calculated tensor contractions for tensor resonances

  auto FIX1 = [&]() {
    MTensor<double> A = MTensor({4, 4, 4, 4, 4, 4}, double(0.0));  // Init with zeros!
    FOR_EACH_6(LI);
    const std::vector<size_t> ind = {u, v, k, l, r, s};
    for (auto const &a1 : LI) {
      for (auto const &s1 : LI) {
        for (auto const &r1 : LI) {
          A(ind) += R_DDDD(u, v, r1, a1) * R_DDDU(k, l, s1, a1) * R_DDUU(r, s, r1, s1);
        }
      }
    }
    FOR_EACH_6_END;
    return A;
  };

  auto FIX2 = [&]() {
    MTensor<double> A = MTensor({4, 4, 4, 4, 4, 4, 4, 4}, double(0.0));  // Init with zeros!
    FOR_EACH_6(LI);
    for (auto const &a1 : LI) {
      for (auto const &r1 : LI) {
        for (auto const &s1 : LI) {
          for (auto const &u1 : LI) {
            A((u, v, k, l, r, s, u1, r1)) +=
                R_DDDD(u, v, u1, a1) * R_DDDU(k, l, s1, a1) * R_DDUU(r, s, r1, s1);
          }
        }
      }
    }
    FOR_EACH_6_END;
    return A;
  };

  auto FIX3 = [&]() {
    MTensor<double> A = MTensor({4, 4, 4, 4, 4, 4, 4, 4}, double(0.0));  // Init with zeros!
    FOR_EACH_6(LI);
    for (auto const &a1 : LI) {
      for (auto const &r1 : LI) {
        for (auto const &s1 : LI) {
          for (auto const &u1 : LI) {
            A((u, v, k, l, r, s, u1, s1)) +=
                R_DDDD(u, v, r1, a1) * R_DDDU(k, l, u1, a1) * R_DDUU(r, s, r1, s1);
          }
        }
      }
    }
    FOR_EACH_6_END;
    return A;
  };

  T1 = FIX1();
  T2 = FIX2();
  T3 = FIX3();
}

}  // namespace gra

```

./src/MFlux.cc 1/2

```
// Photon and other fluxes
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

// C++
#include <complex>
#include <random>
#include <vector>

// Own
#include "Granitti/MFlux.h"

namespace gra {
namespace flux {

using math::abs2;
using math::msqrt;

// Apply non-collinear EPA fluxes at cross section level
// Use with full 2 -> N kinematics
double ApplyktEPAfluxes(double amp2, gra::LORENTZSCALAR& lts) {
  const double gammaflux1 = lts.excite1
                                ? gra::form::IncohFlux(lts.x1, lts.t1, lts.qt1, lts.pfinal[1].M2())
                                : form::CohFlux(lts.x1, lts.t1, lts.qt1);
  const double gammaflux2 = lts.excite2
                                ? gra::form::IncohFlux(lts.x2, lts.t2, lts.qt2, lts.pfinal[2].M2())
                                : form::CohFlux(lts.x2, lts.t2, lts.qt2);

  // const double phasespace = lts.s / lts.s_hat;    // This gives problems at low masses
  const double phasespace = 1.0 / (lts.x1 * lts.x2);  // Consistent with kt-factorization

  // Combine all
  const double fluxes = gammaflux1 * gammaflux2 * phasespace;
  const double tot    = fluxes * amp2;

  // ------------------------------------------------------------------------
  // Apply fluxes to helicity amplitudes
  const double sqrt_fluxes = msqrt(fluxes);  // to "amplitude level"
  for (const auto i : aux::indices(lts.hamp)) { lts.hamp[i] *= sqrt_fluxes; }
  // ------------------------------------------------------------------------

  return tot;
}

// Apply Gamma-Gamma collinear Drees-Zeppenfeld (coherent flux) at cross section level
// Use with collinear kinematics
double ApplyDZfluxes(double amp2, gra::LORENTZSCALAR& lts) {
  // Evaluate gamma pdfs
  const double f1         = form::DZFlux(lts.x1);
  const double f2         = form::DZFlux(lts.x2);
  const double phasespace = 1.0 / (lts.x1 * lts.x2);
  const double tot        = f1 * f2 * amp2 * phasespace;

  return tot;
}

// Apply Gamma-Gamma LUX-pdf (use at \mu > 10 GeV) at cross section level
// Use with collinear kinematics
double ApplyLUXfluxes(double amp2, gra::LORENTZSCALAR& lts) {
  // @@ MULTITHREADING LOCK NEEDED FOR INITIALIZATION @@
  gra::g_mutex.lock();

  // Not created yet
  if (lts.GlobalPdfPtr == nullptr) {
    std::string pdfname = lts.LHAPDFSET;

  retry:
    try {
      lts.GlobalPdfPtr = LHAPDF::mkPDF(pdfname, 0);
      lts.pdf_trials   = 0;  // fine
    } catch (...) {
      ++lts.pdf_trials;
      std::string str = "ApplyLUXfluxes: Problem with reading LHAPDF '" + pdfname + "'";
      aux::AutoDownloadLHAPDF(pdfname);  // Try autodownload
      gra::g_mutex.unlock();             // Remember before throw, otherwise deadlock
      if (lts.pdf_trials >= 2) {         // too many failures
        throw std::invalid_argument(str);
      } else {
        goto retry;
      }
    }
  }
```

./src/MFlux.cc 2/2

```
  gra::g_mutex.unlock();
  // @@ MULTITHREADING UNLOCK @@

  // pdf factorization scale
  const double Q2 = lts.s_hat / 4.0;

  // Evaluate gamma pdfs
  double f1 = 0.0;
  double f2 = 0.0;
  try {
    // Divide x out
    f1 = lts.GlobalPdfPtr->xfxQ2(PDG::PDG_gamma, lts.x1, Q2) / lts.x1;
    f2 = lts.GlobalPdfPtr->xfxQ2(PDG::PDG_gamma, lts.x2, Q2) / lts.x2;
  } catch (...) { throw std::invalid_argument("ApplyLUXfluxes: Failed evaluating LHAPDF"); }

  const double phasespace = 1.0 / (lts.x1 * lts.x2);
  const double tot        = f1 * f2 * amp2 * phasespace;

  return tot;
}

}  // namespace flux
}  // namespace gra
```

```cpp
// GRANIITTI Monte Carlo main class
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

// C++
#include <algorithm>
#include <complex>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <memory>
#include <mutex>
#include <random>
#include <regex>
#include <sstream>
#include <stdexcept>
#include <string>
#include <thread>
#include <vector>

// Own
#include "Graniitti/MAux.h"
#include "Graniitti/MContinuum.h"
#include "Graniitti/MFactorized.h"
#include "Graniitti/MGraniitti.h"
#include "Graniitti/MKinematics.h"
#include "Graniitti/MMath.h"
#include "Graniitti/MNeuroJacobian.h"
#include "Graniitti/MParton.h"
#include "Graniitti/MProcess.h"
#include "Graniitti/MQuasiElastic.h"
#include "Graniitti/MTimer.h"

// HepMC3
#include "HepMC3/GenEvent.h"
#include "HepMC3/WriterAsciiHepMC2.h"
#include "HepMC3/WriterHEPEVT.h"

// Libraries
#include "json.hpp"
#include "rang.hpp"

namespace gra {

// ***************************************************************
// GLOBALS from MGlobals.h: initialized by MGraniitti()

// Model tune
std::string MODELPARAM;

// Multithreading
std::mutex          g_mutex;
std::exception_ptr globalExceptionPtr;

// ***************************************************************

using json = nlohmann::json;

using gra::aux::indices;
using gra::math::PI;
using gra::math::msqrt;
using gra::math::pow2;
using gra::math::pow3;
using gra::math::zi;


// Constructor
MGraniitti::MGraniitti() {
  // ***************************************************************
  // Init program globals

  gra::MODELPARAM         = "null";
  gra::globalExceptionPtr = nullptr;
  // ***************************************************************

  // Print general layout
  PrintInit();
}

// Destructor
MGraniitti::~MGraniitti() {
  // Destroy processes
```



```cpp
  for (std::size_t i = 0; i < pvec.size(); ++i) { delete pvec[i]; }

  std::cout << "~MGraniitti [DONE]" << std::endl;
}

void MGraniitti::PrintHistograms() {
  HistogramFusion();
  proc->PrintHistograms();
}

// Unify histogram bounds across threads, so the histogram fusion is possible
void MGraniitti::UnifyHistogramBounds() {
  // Loop over all 1D histograms
  for (auto const &xpoint : proc->h1) {
    int    xbins = 0;
    double xmin  = 0;
    double xmax  = 0;

    // Fuse buffer values [start index 1]
    for (std::size_t p = 1; p < pvec.size(); ++p) {
      proc->h1[xpoint.first].FuseBuffer(pvec[p]->h1[xpoint.first]);
    }

    proc->h1[xpoint.first].FlushBuffer();
    proc->h1[xpoint.first].GetBounds(xbins, xmin, xmax);
    proc->h1[xpoint.first].ResetBounds(xbins, xmin, xmax);  // New start

    // Loop over processes and set histogram bounds [start index = 1]
    for (std::size_t p = 1; p < pvec.size(); ++p) {
      pvec[p]->h1[xpoint.first].ResetBounds(xbins, xmin, xmax);
    }
  }
  // Loop over all 2D histograms
  for (auto const &xpoint : proc->h2) {
    int    xbins = 0;
    double xmin  = 0;
    double xmax  = 0;

    int    ybins = 0;
    double ymin  = 0;
    double ymax  = 0;

    // Fuse buffer values [start index 1]
    for (std::size_t p = 1; p < pvec.size(); ++p) {
      proc->h2[xpoint.first].FuseBuffer(pvec[p]->h2[xpoint.first]);
    }

    proc->h2[xpoint.first].FlushBuffer();
    proc->h2[xpoint.first].GetBounds(xbins, xmin, xmax, ybins, ymin, ymax);
    proc->h2[xpoint.first].ResetBounds(xbins, xmin, xmax, ybins, ymin, ymax);  // New start

    // Loop over processes and set histogram bounds [start index = 1]
    for (std::size_t p = 1; p < pvec.size(); ++p) {
      pvec[p]->h2[xpoint.first].ResetBounds(xbins, xmin, xmax, ybins, ymin, ymax);
    }
  }
}

// Fuse histograms for N-fold statistics
void MGraniitti::HistogramFusion() {
  if (hist_fusion_done == false) {
    // START with process index 1, because 0 is the base
    for (std::size_t i = 1; i < pvec.size(); ++i) {
      // Loop over all 1D histograms
      for (auto const &xpoint : proc->h1) {
        proc->h1[xpoint.first] = proc->h1[xpoint.first] + pvec[i]->h1[xpoint.first];
      }
      // Loop over all 2D histograms
      for (auto const &xpoint : proc->h2) {
        proc->h2[xpoint.first] = proc->h2[xpoint.first] + pvec[i]->h2[xpoint.first];
      }
    }
    hist_fusion_done = true;
  } else {
    std::cout << "MGraniitti::HistogramFusion: Multithreaded histograms have been "
                 "fused already"
              << std::endl;
  }
}

// Generate events
void MGraniitti::Generate() {
  if (NEVENTS > 0) {  // Generate events
```



```cpp

    // Just before event generation
    // (in order not to generate unnecessary empty files
    // if file name / output type is changed)
    InitFileOutput();
    CallIntegrator(NEVENTS);
  }
}

// This is called just before event generation
// Check against nullptr is done below, because if the output is already set
// externally,
// this function is not used.
void MGraniitti::InitFileOutput() {
  if (NEVENTS > 0) {
    if (OUTPUT == "") {  // OUTPUT must be set
      throw std::invalid_argument("MGraniitti::InitFileOutput: OUTPUT filename not set!");
    }

    FULL_OUTPUT_STR = gra::aux::GetBasePath(2) + "/output/" + OUTPUT + "." + FORMAT;

    // ---------------------------------------------------------------
    // Generator info
    runinfo = std::make_shared<HepMC3::GenRunInfo>();

    struct HepMC3::GenRunInfo::ToolInfo generator = {
        std::string("GRANIITTI (" + gra::MODELPARAM + ")"),
        std::to_string(aux::GetVersion()).substr(0, 5), std::string("Generator")};
    runinfo->tools().push_back(generator);

    struct HepMC3::GenRunInfo::ToolInfo config = {FULL_INPUT_STR, "1.0",
                                                  std::string("Steering card")};
    runinfo->tools().push_back(config);

    // ---------------------------------------------------------------

    if (FORMAT == "hepmc3" && outputHepMC3 == nullptr) {
      outputHepMC3 = std::make_shared<HepMC3::WriterAscii>(FULL_OUTPUT_STR, runinfo);
    } else if (FORMAT == "hepmc2" && outputHepMC2 == nullptr) {
      outputHepMC2 = std::make_shared<HepMC3::WriterAsciiHepMC2>(FULL_OUTPUT_STR, runinfo);
    } else if (FORMAT == "hepevt" && outputHEPEVT == nullptr) {
      outputHEPEVT = std::make_shared<HepMC3::WriterHEPEVT>(FULL_OUTPUT_STR);
    }
  }
}

// Return process numbers available
std::vector<std::string> MGraniitti::GetProcessNumbers() const {
  std::cout << rang::style::bold;
  std::cout << "Available processes: ";
  std::cout << rang::style::reset << std::endl << std::endl;
  std::cout << std::endl;

  std::cout << rang::style::bold << "2->3 x 1->(N-2) LIPS:" << rang::style::reset << std::endl;
  std::vector<std::string> list1 = proc_F.ProcPtr.PrintProcesses();
  std::cout << std::endl;

  std::cout << rang::style::bold << "2->N LIPS:" << rang::style::reset << std::endl;
  std::vector<std::string> list2 = proc_C.ProcPtr.PrintProcesses();
  std::cout << std::endl;

  std::cout << rang::style::bold << "Quasielastic:" << rang::style::reset << std::endl;
  std::vector<std::string> list3 = proc_Q.ProcPtr.PrintProcesses();
  std::cout << std::endl;

  std::cout << rang::style::bold << "2->1 x (1->M) collinear:" << rang::style::reset << std::endl;
  std::vector<std::string> list4 = proc_F.ProcPtr.PrintProcesses();
  std::cout << std::endl;

  // Concatenate all
  list1.insert(list1.end(), list2.begin(), list2.end());
  list1.insert(list1.end(), list3.begin(), list3.end());
  list1.insert(list1.end(), list4.begin(), list4.end());

  return list1;
}

// (Re-)assign the pointers to the local memory space
void MGraniitti::InitProcessMemory(std::string process, unsigned int seed) {
  // These must be here!
  PROCESS = process;

  // <Q> process
```



```cpp
  if (proc_Q.ProcPtr.ProcessExist(process)) {
    proc_Q = MQuasiElastic(process, syntax);
    proc   = &proc_Q;

    // <F> processes
  } else if (proc_F.ProcPtr.ProcessExist(process)) {
    proc_F = MFactorized(process, syntax);
    proc   = &proc_F;

    // <C> processes
  } else if (proc_C.ProcPtr.ProcessExist(process)) {
    proc_C = MContinuum(process, syntax);
    proc   = &proc_C;

    // <P> processes
  } else if (proc_P.ProcPtr.ProcessExist(process)) {
    proc_P = MParton(process, syntax);
    proc   = &proc_P;
  } else {
    std::string str = "MGraniitti::InitProcessMemory: Unknown PROCESS: " + process;
    // GetProcessNumbers(); // This is done by main program
    throw std::invalid_argument(str);
  }

  // Set random seed last!
  proc->random.SetSeed(seed);
}

void MGraniitti::InitMultiMemory() {
  // ** Init multithreading memory by making copies of the process **
  for (std::size_t i = 0; i < pvec.size(); ++i) { delete pvec[i]; }
  pvec.resize(CORES, nullptr);

  // Create new process objects for each thread
  for (int i = 0; i < CORES; ++i) {
    if (proc_Q.ProcPtr.ProcessExist(PROCESS)) {
      pvec[i] = new MQuasiElastic(proc_Q);
    } else if (proc_F.ProcPtr.ProcessExist(PROCESS)) {
      pvec[i] = new MFactorized(proc_F);
    } else if (proc_C.ProcPtr.ProcessExist(PROCESS)) {
      pvec[i] = new MContinuum(proc_C);
    } else if (proc_P.ProcPtr.ProcessExist(PROCESS)) {
      pvec[i] = new MParton(proc_P);
    }
  }

  // RANDOM SEED PER THREAD (IMPORTANT!)
  for (int i = 0; i < CORES; ++i) {
    const unsigned int tid = i + 1;
    // Deterministic seed
    const unsigned int thread_seed =
        static_cast<unsigned int>(std::max(0, (int)pvec[i]->random.GetSeed()) * tid);
    pvec[i]->random.SetSeed(thread_seed);
  }

  // SET main control pointer to the first one
  // (needed for printing etc.)
  proc = pvec[0];
  proc->PrintInit(HILJAA);
}

// Set simple MC parameters
void MGraniitti::SetMCParam(MCPARAM &in) { mcparam = in; }

// Set VEGAS parameters
void MGraniitti::SetVegasParam(const VEGASPARAM &in) { vparam = in; }

// Read parameters from a single JSON file
void MGraniitti::ReadInput(const std::string &inputfile, const std::string cmd_PROCESS) {
  // Save it for later use
  FULL_INPUT_STR = inputfile;

  std::cout << rang::fg::green << "MGraniitti::ReadInput: " + inputfile << rang::fg::reset
            << std::endl
            << std::endl;

  ReadGeneralParam(inputfile);
  ReadProcessParam(inputfile, cmd_PROCESS);
  ReadModelParam(gra::MODELPARAM);
}

// General parameter initialization
void MGraniitti::ReadGeneralParam(const std::string &inputfile) {
```

```
333:   // Read and parse
334:   const std::string data = gra::aux::GetInputData(inputfile);
335:   json              j;
336:   try {
337:     j = json::parse(data);
338:   } catch (...) {
339:     std::string str = "MGraniitti::ReadGeneralParam: Error parsing " + inputfile +
340:                       " (Check for extra/missing commas)";
341:     throw std::invalid_argument(str);
342:   }
343:
344:   // JSON block identifier
345:   const std::string XID = "GENERALPARAM";
346:
347:   // Setup parameters (order is important)
348:   SetNumberOfEvents(j.at(XID).at("NEVENTS"));
349:   SetOutput(j.at(XID).at("OUTPUT"));
350:   SetFormat(j.at(XID).at("FORMAT"));
351:   SetWeighted(j.at(XID).at("WEIGHTED"));
352:   SetIntegrator(j.at(XID).at("INTEGRATOR"));
353:   SetCores(j.at(XID).at("CORES"));
354:
355:   // Save for later use
356:   gra::MODELPARAM = j.at(XID).at("MODELPARAM");
357: }
358:
359: // General model parameters initialized from .json file
360: //
361: void MGraniitti::ReadModelParam(const std::string &inputfile) const {
362:   const std::string fullpath =
363:       gra::aux::GetBasePath(2) + "/modeldata/" + inputfile + "/GENERAL.json";
364:
365:   // Read generic blocks
366:   PARAM_SOFT::ReadParameters(fullpath);
367:   PARAM_STRUCTURE::ReadParameters(fullpath);
368:   PARAM_FLAT::ReadParameters(fullpath);
369:   PARAM_NSTAR::ReadParameters(fullpath);
370:
371:   // The rest are handled by spesific amplitude classes
372: }
373:
374: // Process parameter initialization, Call proc->post_Constructor() after this
375: void MGraniitti::ReadProcessParam(const std::string &inputfile, const std::string cmd_PROCESS) {
376:   // Read and parse
377:   const std::string data = gra::aux::GetInputData(inputfile);
378:   json              j;
379:   try {
380:     j = json::parse(data);
381:   } catch (...) {
382:     std::string str = "MGraniitti::ReadProcessParam: Error parsing " + inputfile +
383:                       " (Check for extra/missing commas)";
384:     throw std::invalid_argument(str);
385:   }
386:   const std::string XID = "PROCESSPARAM";
387:
388:   // ----------------------------------------------------------------
389:   // Initialize process
390:
391:   std::string fullstring;
392:   if (cmd_PROCESS == "null") {
393:     fullstring = j.at(XID).at("PROCESS");
394:   } else {  // commandline override
395:     fullstring = cmd_PROCESS;
396:   }
397:
398:   // ----------------------------------------------------------------
399:   // First separate possible extra arguments by @... ...
400:   std::vector<std::size_t> markerpos = aux::FindOccurance(fullstring, "@");
401:
402:   if (markerpos.size() != 0) {
403:     syntax     = aux::SplitCommands(fullstring.substr(markerpos[0]));
404:     fullstring = fullstring.substr(0, markerpos[0] - 1);
405:   }
406:   // ----------------------------------------------------------------
407:
408:   // Now separate process and decay parts by "->"
409:   std::string PROCESS_PART = "";
410:   std::string DECAY_PART   = "";
411:   std::size_t pos          = 0;
412:   std::size_t pos1         = fullstring.find("->");
413:   std::size_t pos2         = fullstring.find("&>");  // ISOLATEd Phase-Space
414:
415:   pos = (pos1 != std::string::npos) ? pos1 : std::string::npos;  // Try to find ->
```



```
416:   if (pos == std::string::npos) {
417:     pos = (pos2 != std::string::npos) ? pos2 : std::string::npos;  // Try to find &>
418:   }
419:
420:   if (pos != std::string::npos) {
421:     PROCESS_PART = fullstring.substr(0, pos - 1);  // beginning to the pos-1
422:     DECAY_PART   = fullstring.substr(pos + 2);     // from pos+2 to the end
423:   } else {
424:     PROCESS_PART = fullstring;  // No decay defined
425:   }
426:
427:   // Trim extra spaces away
428:   gra::aux::TrimExtraSpace(PROCESS_PART);
429:   gra::aux::TrimExtraSpace(DECAY_PART);
430:
431:   InitProcessMemory(PROCESS_PART, j.at(XID).at("RNDSEED"));
432:
433:   // ----------------------------------------------------------------
434:   // SETUP process: process memory needs to be initialized before!
435:
436:   proc->SetLHAPDF(j.at(XID).at("LHAPDF"));
437:   proc->SetScreening(j.at(XID).at("POMLOOP"));
438:   proc->SetExcitation(j.at(XID).at("NSTARS"));
439:   proc->SetHistograms(j.at(XID).at("HIST"));
440:
441:   if (pos2 != std::string::npos) {
442:     if (PROCESS_PART.find("<P>") == std::string::npos &&
443:         PROCESS_PART.find("<P>") == std::string::npos) {
444:       throw std::invalid_argument(
445:           "MGraniitti::ReadProcessParam: Phase space "
446:           "isolation arrow '&>' to be "
447:           "used only with <P> or <P> class!");
448:     }
449:     proc->SetISOLATE(true);
450:   }
451:
452:   // ----------------------------------------------------------------
453:   // Decaymode setup
454:   proc->SetDecayMode(DECAY_PART);
455:
456:   // Read free resonance parameters (only if needed)
457:   std::map<std::string, gra::PARAM_RES> RESONANCES;
458:   if (PROCESS_PART.find("RES") != std::string::npos) {
459:     // From .json input
460:     std::vector<std::string> RES = j.at(XID).at("RES");
461:
462:     // @ Command syntax override
463:     for (const auto &i : indices(syntax)) {
464:       if (syntax[i].id == "RES") {
465:         std::vector<std::string> temp;
466:         for (const auto &x : syntax[i].arg) {
467:           if (x.second == "true" || x.second == "1") {
468:             temp.push_back(x.first);  // f0_980, ...
469:           }
470:         }
471:         RES = temp;
472:       }
473:     }
474:
475:     // Read resonance data
476:     for (std::size_t i = 0; i < RES.size(); ++i) {
477:       const std::string str = "RES_" + RES[i] + ".json";
478:       RESONANCES[RES[i]]    = gra::form::ReadResonance(str, proc->random);
479:     }
480:
481:     // @ Command syntax override "on-the-flight parameters"
482:     for (const auto &i : indices(syntax)) {
483:       if (syntax[i].id == "R") {
484:         // Take target string
485:         std::string RESNAME;
486:         if (syntax[i].target.size() == 1) {
487:           RESNAME = syntax[i].target[0];
488:         } else if (syntax[i].target.size() == 0) {
489:           throw std::invalid_argument("@Syntax error: invalid R[] without any target []");
490:         } else if (syntax[i].target.size() > 1) {
491:           throw std::invalid_argument("@Syntax error: invalid R[] with multiple targets inside []");
492:         }
493:
494:         // Do we find target
495:         if (RESONANCES.find(RESNAME) == RESONANCES.end()) {
496:           throw std::invalid_argument("@Syntax error: invalid R[" + RESNAME + "] not found");
497:         }
498:         bool couplings_touched = false;
```



```
499:         bool density_touched   = false;
500:
501:         MMatrix<std::complex<double>> newrho(1, 1, 0.0);
502:         if (RESONANCES[RESNAME].p.spinX2 != 0) {
503:           const int N = RESONANCES[RESNAME].rho.size_row();
504:           newrho      = MMatrix<std::complex<double>>(N, N, 0.0);
505:         }
506:
507:         // Loop over key:val arguments
508:         for (const auto &x : syntax[i].arg) {
509:           if (x.first == "M") {
510:             RESONANCES[RESNAME].p.mass = std::stod(x.second);
511:             std::cout << rang::fg::green << "@R[" << RESNAME
512:                       << "] new mass: " << RESONANCES[RESNAME].p.mass << rang::fg::reset
513:                       << std::endl;
514:           }
515:           if (x.first == "W") {
516:             RESONANCES[RESNAME].p.width = std::stod(x.second);
517:             std::cout << rang::fg::green << "@R[" << RESNAME
518:                       << "] new width: " << RESONANCES[RESNAME].p.width << rang::fg::reset
519:                       << std::endl;
520:           }
521:
522:           // Set couplings
523:           for (std::size_t n = 0; n < RESONANCES[RESNAME].g_Tensor.size(); ++n) {
524:             if (x.first == ("g" + std::to_string(n))) {
525:               RESONANCES[RESNAME].g_Tensor[n] = std::stod(x.second);
526:               couplings_touched               = true;
527:             }
528:           }
529:
530:           // Set diagonal spin density elements
531:           for (std::size_t n = 0; n <= (newrho.size_row() - 1) / 2; ++n) {
532:             const int J = (newrho.size_row() - 1) / 2;
533:             if (x.first == ("JZ" + std::to_string(n))) {
534:               const double value = std::stod(x.second);
535:
536:               if (value < 0) {
537:                 throw std::invalid_argument("MGraniitti::ReadProcessParam: @R[] JZ value < 0");
538:               }
539:
540:               newrho(newrho.size_row() - 1 - J - n)[newrho.size_row() - 1 - J - n] = value;
541:
542:               newrho(newrho.size_row() - 1 - J + n)[newrho.size_row() - 1 - J + n] = value;
543:               density_touched                                                     = true;
544:             }
545:           }
546:         }
547:
548:         // Normalize the trace and save it
549:         if (density_touched) {
550:           const std::complex<double> trace = newrho.Trace();
551:           if (std::abs(trace) > 0) {
552:             newrho = newrho * (1.0 / trace);
553:           } else {
554:             throw std::invalid_argument(
555:                 "MGraniitti::ReadProcessParam: @R[] Spin density error with <= 0 trace");
556:           }
557:           RESONANCES[RESNAME].rho = newrho;
558:
559:           std::cout << rang::fg::green << "@R[" << RESNAME
560:                     << "] new spin density set:" << std::endl;
561:           RESONANCES[RESNAME].rho.Print();
562:           std::cout << rang::fg::reset << std::endl;
563:         }
564:
565:         // Print new coupling array
566:         if (couplings_touched) {
567:           std::cout << rang::fg::green << "@R[" << RESNAME
568:                     << "] new production couplings set: g_Tensor = [";
569:           for (std::size_t k = 0; k < RESONANCES[RESNAME].g_Tensor.size(); ++k) {
570:             printf("%0.3E", RESONANCES[RESNAME].g_Tensor[k]);
571:             if (k < RESONANCES[RESNAME].g_Tensor.size() - 1) { std::cout << ", "; }
572:           }
573:           std::cout << "]" << rang::fg::reset << std::endl;
574:         }
575:       }
576:     }
577:     proc->SetResonances(RESONANCES);
578:
579:     // Setup resonance branching (final step!)
580:     proc->SetupBranching();
581:   }
```



```
582:   // ----------------------------------------------------------------
583:
584:   // Command syntax parameters
585:   for (const auto &i : indices(syntax)) {
586:     if (syntax[i].id == "FLATAMP") { proc->SetFLATAMP(std::stoi(syntax[i].arg["_SINGLET_"])); }
587:     if (syntax[i].id == "FLATMASS2") { proc->SetFLATMASS2(syntax[i].arg["_SINGLET_"] == "true"); }
588:     if (syntax[i].id == "OFFSHELL") { proc->SetOFFSHELL(std::stod(syntax[i].arg["_SINGLET_"])); }
589:
590:     if (syntax[i].id == "SPINGEN") { proc->SetSPINGEN(syntax[i].arg["_SINGLET_"] == "true"); }
591:     if (syntax[i].id == "SPINDEC") { proc->SetSPINDEC(syntax[i].arg["_SINGLET_"] == "true"); }
592:
593:     if (syntax[i].id == "FRAME") { proc->SetFRAME(syntax[i].arg["_SINGLET_"]); }
594:     if (syntax[i].id == "JMAX") { proc->SetJMAX(std::stoi(syntax[i].arg["_SINGLET_"])); }
595:   }
596:
597:   // Always last (we need PDG tables)
598:   std::vector<std::string> beam   = j.at(XID).at("BEAM");
599:   std::vector<double>      energy = j.at(XID).at("ENERGY");
600:   proc->SetInitialState(beam, energy);
601:
602:   // Now rest of the parameters
603:   ReadIntegralParam(inputfile);
604:   ReadGenCuts(inputfile);
605:   ReadFidCuts(inputfile);
606:   ReadVetoCuts(inputfile);
607: }
608:
609: // MC integrator parameter initialization
610: void MGraniitti::ReadIntegralParam(const std::string &inputfile) {
611:   using namespace gra::aux;
612:
613:   // Read and parse
614:   const std::string data = gra::aux::GetInputData(inputfile);
615:   json              j;
616:   try {
617:     j = json::parse(data);
618:   } catch (...) {
619:     std::string str = "MGraniitti::ReadIntegralParam: Error parsing " + inputfile +
620:                       " (Check for extra/missing commas)";
621:     throw std::invalid_argument(str);
622:   }
623:   const std::string XID = "INTEGRALPARAM";
624:
625:   // Numerical loop integration
626:   const int ND = j.at(XID).at("POMLOOP").at("ND");
627:   AssertRange(ND, {-10, 10}, "POMLOOP::ND", true);
628:   proc->Eikonal.Numerics.SetLoopDiscretization(ND);
629:
630:   // FLAT (naive) MC parameters
631:   MCPARAM mpam;
632:   mpam.PRECISION = j.at(XID).at("FLAT").at("PRECISION");
633:   AssertRange(mpam.PRECISION, {0.0, 1.0), "FLAT::PRECISION", true);
634:   mpam.MIN_EVENTS = j.at(XID).at("FLAT").at("MIN_EVENTS");
635:   AssertRange(mpam.MIN_EVENTS, {10, (unsigned int)1e9), "FLAT::MIN_EVENTS", true);
636:   SetMCParam(mpam);
637:
638:   try {
639:     j.at(XID).at("VEGAS").at("BINS");
640:   } catch (...) {
641:     std::cout << "MGraniitti::ReadIntegralParam: Did not found VEGAS parameter block "
642:                  "from json input, using default"
643:               << std::endl;
644:     return;  // Did not find VEGAS parameter block at all, use default
645:   }
646:
647:   // VEGAS parameters
648:   VEGASPARAM vpam;
649:   vpam.BINS = j.at(XID).at("VEGAS").at("BINS");
650:   AssertRange(vpam.BINS, {0, (unsigned int)1e9), "VEGAS::BINS", true);
651:   if ((vpam.BINS % 2) != 0) {
652:     throw std::invalid_argument("VEGAS::BINS = " + std::to_string(vpam.BINS) +
653:                                 " should be even number");
654:   }
655:
656:   vpam.LAMBDA = j.at(XID).at("VEGAS").at("LAMBDA");
657:   AssertRange(vpam.LAMBDA, {1.0, 10.0), "VEGAS::LAMBDA", true);
658:   vpam.NCALL = j.at(XID).at("VEGAS").at("NCALL");
659:   AssertRange(vpam.NCALL, {50, (unsigned int)1e9), "VEGAS::NCALL", true);
660:   vpam.ITER = j.at(XID).at("VEGAS").at("ITER");
661:   AssertRange(vpam.ITER, {1, (unsigned int)1e9), "VEGAS::ITER", true);
662:   vpam.CHI2MAX = j.at(XID).at("VEGAS").at("CHI2MAX");
663:   AssertRange(vpam.CHI2MAX, {0.0, 1e3), "VEGAS::CHI2MAX", true);
664:   vpam.PRECISION = j.at(XID).at("VEGAS").at("PRECISION");
```

```
665:   AssertRange(vpam.PRECISION, {0.0, 1.0}, "VEGAS::PRECISION", true);
666:   vpam.DEBUG = j.at(XID).at("VEGAS").at("DEBUG");
667:   AssertSet(vpam.DEBUG, {-1, 0, 1}, "VEGAS::DEBUG", true);
668:
669:   SetVegasParam(vpam);
670: }
671:
672: // Generator cuts
673: void MGraniitti::ReadGenCuts(const std::string &inputfile) {
674:   // Read and parse
675:   const std::string data = gra::aux::GetInputData(inputfile);
676:   json              j;
677:   try {
678:     j = json::parse(data);
679:   } catch (...) {
680:     std::string str =
681:         "MGraniitti::ReadGenCuts: Error parsing " + inputfile + " (Check for extra/missing commas)";
682:     throw std::invalid_argument(str);
683:   }
684:   const std::string XID = "GENCUTS";
685:
686:   gra::GENCUT gcuts;
687:
688:   // Continuum phase space class
689:   if (PROCESS.find("<C>") != std::string::npos) {
690:     // Daughter rapidity
691:     std::vector<double> rap;
692:     try {
693:       std::vector<double> temp = j.at(XID).at("<C>").at("Rap");
694:       rap                      = temp;
695:     } catch (...) {
696:       throw std::invalid_argument(
697:           "MGraniitti::ReadGenCuts: <C> phase space class requires from user: "
698:           "\"GENCUTS\" : { \"<C>\" : { \"Rap\" : [min, max] }}");
699:     }
700:     gcuts.rap_min = rap[0];
701:     gcuts.rap_max = rap[1];
702:     gra::aux::AssertCut(rap, "GENCUTS::<C>::Rap", true);
703:
704:     // This is optional, intermediate kt
705:     std::vector<double> kt;
706:     try {
707:       std::vector<double> temp = j.at(XID).at("<C>").at("Kt");
708:       kt                       = temp;
709:     } catch (...) {
710:       // Do nothing
711:     }
712:     if (kt.size() != 0) {
713:       gcuts.kt_min = kt[0];
714:       gcuts.kt_max = kt[1];
715:       gra::aux::AssertCutRange(kt, {0.0, 1e32}, "GENCUTS::<C>::Kt", true);
716:     }
717:
718:     // This is optional, forward leg pt
719:     std::vector<double> pt;
720:     try {
721:       std::vector<double> temp = j.at(XID).at("<C>").at("Pt");
722:       pt                       = temp;
723:     } catch (...) {
724:       // Do nothing
725:     }
726:     if (pt.size() != 0) {
727:       gcuts.forward_pt_min = pt[0];
728:       gcuts.forward_pt_max = pt[1];
729:       gra::aux::AssertCutRange(pt, {0.0, 1e32}, "GENCUTS::<C>::Pt", true);
730:     }
731:
732:     // This is optional, forward leg excitation
733:     std::vector<double> Xi;
734:     try {
735:       std::vector<double> temp = j.at(XID).at("<C>").at("Xi");
736:       Xi                       = temp;
737:     } catch (...) {
738:       // Do nothing
739:     }
740:     if (Xi.size() != 0) {
741:       gcuts.Xi_min = Xi[0];
742:       gcuts.Xi_max = Xi[1];
743:       gra::aux::AssertCutRange(Xi, {0.0, 1.0}, "GENCUTS::<C>::Xi", true);
744:     }
745:   }
746:
747:   // Factorized phase space class
```



```
748:   if (PROCESS.find("<F>") != std::string::npos) {
749:     // System rapidity
750:     std::vector<double> Y;
751:     try {
752:       std::vector<double> temp = j.at(XID).at("<F>").at("Rap");
753:       Y                        = temp;
754:     } catch (...) {
755:       throw std::invalid_argument(
756:           "MGraniitti::ReadGenCuts: <F> phase space class requires from user: "
757:           "\"GENCUTS\" : { \"<F>\" : { \"Rap\" : [min, max] }}");
758:     }
759:     gcuts.Y_min = Y[0];
760:     gcuts.Y_max = Y[1];
761:     gra::aux::AssertCut(Y, "GENCUTS::<F>::Rap", true);
762:
763:     // System mass
764:     std::vector<double> M;
765:     try {
766:       std::vector<double> temp = j.at(XID).at("<F>").at("M");
767:       M                        = temp;
768:     } catch (...) {
769:       throw std::invalid_argument(
770:           "MGraniitti::ReadGenCuts: <F> phase space class requires from user: "
771:           "\"GENCUTS\" : { \"<F>\" : { \"M\" : [min, max] }}");
772:     }
773:     gcuts.M_min = M[0];
774:     gcuts.M_max = M[1];
775:     gra::aux::AssertCutRange(M, {0.0, 1e32}, "GENCUTS::<F>::M", true);
776:
777:     // This is optional, forward leg pt
778:     std::vector<double> pt;
779:     try {
780:       std::vector<double> temp = j.at(XID).at("<F>").at("Pt");
781:       pt                       = temp;
782:     } catch (...) {
783:       // Do nothing
784:     }
785:     if (pt.size() != 0) {
786:       gcuts.forward_pt_min = pt[0];
787:       gcuts.forward_pt_max = pt[1];
788:       gra::aux::AssertCutRange(pt, {0.0, 1e32}, "GENCUTS::<F>::Pt", true);
789:     }
790:
791:     // This is optional, forward leg excitation
792:     std::vector<double> Xi;
793:     try {
794:       std::vector<double> temp = j.at(XID).at("<F>").at("Xi");
795:       Xi                       = temp;
796:     } catch (...) {
797:       // Do nothing
798:     }
799:     if (Xi.size() != 0) {
800:       gcuts.Xi_min = Xi[0];
801:       gcuts.Xi_max = Xi[1];
802:       gra::aux::AssertCutRange(Xi, {0.0, 1.0}, "GENCUTS::<F>::Xi", true);
803:     }
804:   }
805:
806:   // Quasielastic phase space class
807:   if (PROCESS.find("<Q>") != std::string::npos) {
808:     std::vector<double> Xi;
809:     try {
810:       std::vector<double> temp = j.at(XID).at("<Q>").at("Xi");
811:       Xi                       = temp;
812:     } catch (...) {
813:       throw std::invalid_argument(
814:           "MGraniitti::ReadGenCuts: <Q> phase space class requires from user: "
815:           "\"GENCUTS\" : { \"<Q>\" : { \"Xi\" : [min, max] }}");
816:     }
817:     gcuts.XI_min = Xi[0];
818:     gcuts.XI_max = Xi[1];
819:     gra::aux::AssertCutRange(Xi, {0.0, 1.0}, "GENCUTS::<Q>::Xi", true);
820:   }
821:
822:   proc->SetGenCuts(gcuts);
823: }
824:
825: // Fiducial cuts
826: void MGraniitti::ReadFidCuts(const std::string &inputfile) {
827:   // Read and parse
828:   const std::string data = gra::aux::GetInputData(inputfile);
829:   json              j;
830:   try {
```



```
831:     j = json::parse(data);
832:   } catch (...) {
833:     std::string str =
834:         "MGraniitti::ReadFidCuts: Error parsing " + inputfile + " (Check for extra/missing commas)";
835:     throw std::invalid_argument(str);
836:   }
837:
838:   // Fiducial cuts
839:   gra::FIDCUT       fcuts;
840:   const std::string XID = "FIDCUTS";
841:
842:   try {
843:     fcuts.active = j.at(XID).at("active");
844:   } catch (...) {
845:     return;  // Did not find cut block at all
846:   }
847:
848:   // Particles
849:   {
850:     std::vector<double> eta = j.at(XID).at("PARTICLE").at("Eta");
851:     fcuts.eta_min           = eta[0];
852:     fcuts.eta_max           = eta[1];
853:     gra::aux::AssertCut(eta, "FIDCUTS::PARTICLE::Eta", true);
854:
855:     std::vector<double> rap = j.at(XID).at("PARTICLE").at("Rap");
856:     fcuts.rap_min           = rap[0];
857:     fcuts.rap_max           = rap[1];
858:     gra::aux::AssertCut(rap, "FIDCUTS::PARTICLE::Rap", true);
859:
860:     std::vector<double> pt = j.at(XID).at("PARTICLE").at("Pt");
861:     fcuts.pt_min           = pt[0];
862:     fcuts.pt_max           = pt[1];
863:     gra::aux::AssertCut(pt, "FIDCUTS::PARTICLE::Pt", true);
864:
865:     std::vector<double> Et = j.at(XID).at("PARTICLE").at("Et");
866:     fcuts.Et_min           = Et[0];
867:     fcuts.Et_max           = Et[1];
868:     gra::aux::AssertCut(Et, "FIDCUTS::PARTICLE::Et", true);
869:   }
870:
871:   // System
872:   {
873:     std::vector<double> M = j.at(XID).at("SYSTEM").at("M");
874:     fcuts.M_min           = M[0];
875:     fcuts.M_max           = M[1];
876:     gra::aux::AssertCut(M, "FIDCUTS::SYSTEM::M", true);
877:
878:     std::vector<double> Y = j.at(XID).at("SYSTEM").at("Rap");
879:     fcuts.Y_min           = Y[0];
880:     fcuts.Y_max           = Y[1];
881:     gra::aux::AssertCut(Y, "FIDCUTS::SYSTEM::Rap", true);
882:
883:     std::vector<double> Pt = j.at(XID).at("SYSTEM").at("Pt");
884:     fcuts.Pt_min           = Pt[0];
885:     fcuts.Pt_max           = Pt[1];
886:     gra::aux::AssertCut(Pt, "FIDCUTS::SYSTEM::Pt", true);
887:   }
888:
889:   // Forward
890:   {
891:     std::vector<double> t = j.at(XID).at("FORWARD").at("t");
892:     fcuts.forward_t_min   = t[0];
893:     fcuts.forward_t_max   = t[1];
894:     gra::aux::AssertCut(t, "FIDCUTS::FORWARD::t", true);
895:
896:     std::vector<double> M = j.at(XID).at("FORWARD").at("M");
897:     fcuts.forward_M_min   = M[0];
898:     fcuts.forward_M_max   = M[1];
899:     gra::aux::AssertCut(M, "FIDCUTS::FORWARD::M", true);
900:   }
901:
902:   // Set fiducial cuts
903:   proc->SetFidCuts(fcuts);
904:
905:   // Set user cuts
906:   proc->SetUserCuts(j.at(XID).at("USERCUTS"));
907: }
908:
909: // Fiducial cuts
910: void MGraniitti::ReadVetoCuts(const std::string &inputfile) {
911:   // Read and parse
912:   const std::string data = gra::aux::GetInputData(inputfile);
913:   json              j;
```



```
914:   try {
915:     j = json::parse(data);
916:   } catch (...) {
917:     std::string str = "MGraniitti::ReadVetoCuts: Error parsing " + inputfile +
918:                       " (Check for extra/missing commas)";
919:     throw std::invalid_argument(str);
920:   }
921:
922:   // Fiducial cuts
923:   gra::VETOCUT      veto;
924:   const std::string XID = "VETOCUTS";
925:
926:   try {
927:     veto.active = j.at(XID).at("active");
928:   } catch (...) {
929:     return;  // Did not find cut block at all
930:   }
931:
932:   // Find domains
933:   for (std::size_t i = 0; i < 100; ++i) {
934:     const std::string NUMBER = std::to_string(i);
935:
936:     gra::VETODOMAIN     domain;
937:     std::vector<double> eta;
938:     std::vector<double> pt;
939:
940:     // try to find new domain
941:     try {
942:       std::vector<double> temp1 = j.at(XID).at(NUMBER).at("Eta");
943:       eta                       = temp1;
944:       std::vector<double> temp2 = j.at(XID).at(NUMBER).at("Pt");
945:       pt                        = temp2;
946:     } catch (...) {
947:       break;  // Not found
948:     }
949:
950:     domain.eta_min = eta[0];
951:     domain.eta_max = eta[1];
952:     gra::aux::AssertCut(eta, "VETOCUTS::" + NUMBER + "::Eta", true);
953:
954:     domain.pt_min = pt[0];
955:     domain.pt_max = pt[1];
956:     gra::aux::AssertCut(pt, "VETOCUTS::" + NUMBER + "::Pt", true);
957:
958:     veto.cuts.push_back(domain);
959:   }
960:
961:   // Set fiducial cuts
962:   proc->SetVetoCuts(veto);
963: }
964:
965: // Get maximum vegasweight
966: double MGraniitti::GetMaxweight() const {
967:   if (INTEGRATOR == "VEGAS") {
968:     return stat.maxf;
969:   } else {
970:     return stat.maxW;
971:   }
972: }
973:
974: // Set maximum weight for the integration process
975: void MGraniitti::SetMaxweight(double weight) {
976:   if (weight > 0) {
977:     if (INTEGRATOR == "VEGAS") {
978:       stat.maxf = weight;
979:     } else {
980:       stat.maxW = weight;
981:     }
982:   } else {
983:     std::string str =
984:         "MGraniitti::SetMaxweight: Maximum weight: " + std::to_string(weight) + " not valid!";
985:     throw std::invalid_argument(str);
986:   }
987: }
988:
989: void MGraniitti::PrintInit() const {
990:   if (!HILJAA) {
991:     gra::aux::PrintFlashScreen(rang::fg::yellow);
992:     std::cout << rang::style::bold
993:               << "GRANIITTI - Monte Carlo event generator for "
994:                  "high energy diffraction"
995:               << rang::style::reset << std::endl
996:               << std::endl;
```

```
  997:     gra::aux::PrintVersion();
  998:     gra::aux::PrintBar("-");
  999:
 1000:     const double GB = pow3(1024.0);
 1001:     printf("Running on %s (%d CORE / %0.2f GB RAM) at %s \n", gra::aux::HostName().c_str(),
 1002:            std::thread::hardware_concurrency(), gra::aux::TotalSystemMemory() / GB,
 1003:            gra::aux::DateTime().c_str());
 1004:     int64_t size = 0;
 1005:     int64_t free = 0;
 1006:     int64_t used = 0;
 1007:     gra::aux::GetDiskUsage("/", size, free, used);
 1008:     printf("Path '/': size | used | free = %0.1f | %0.1f | %0.1f GB \n", size / GB, used / GB,
 1009:            free / GB);
 1010:     std::cout << "Program path: " << gra::aux::GetBasePath(2) << std::endl;
 1011:     std::cout << gra::aux::SystemName() << std::endl;
 1012:     gra::aux::PrintBar("-");
 1013:   }
 1014: }
 1015:
 1016: void MGraniitti::Initialize() {
 1017:   // ** ALWAYS HERE ONLY AS LAST! **
 1018:   proc->post_Constructor();
 1019:
 1020:   // Print out basic information
 1021:   std::cout << std::endl;
 1022:   std::cout << rang::style::bold << "General setup:" << rang::style::reset << std::endl
 1023:             << std::endl;
 1024:   std::cout << "Output file:              " << OUTPUT << std::endl;
 1025:   std::cout << "Output format:            " << FORMAT << std::endl;
 1026:   std::cout << "Multithreading:           " << CORES << std::endl;
 1027:   std::cout << "Integrator:               " << INTEGRATOR << std::endl;
 1028:   std::cout << "Number of events:         " << NEVENTS << std::endl;
 1029:   std::cout << "Parameter setup:          " << gra::MODELPARAM << std::endl;
 1030:
 1031:   std::string str = (WEIGHTED == true) ? "weighted" : "unweighted";
 1032:   std::cout << rang::fg::green << "Generation mode:          " << str << rang::fg::reset << std::endl;
 1033:   std::cout << std::endl;
 1034:
 1035:   // If Eikonals are not yet initialized and pomeron screening loop is on
 1036:   if (proc->Eikonal.IsInitialized() == false && proc->GetScreening() == true) {
 1037:     proc->Eikonal.S3Constructor(proc->GetMandelstam_s(), proc->GetInitialState(), false);
 1038:   }
 1039:
 1040:   // Integrate
 1041:   CallIntegrator(0);
 1042: }
 1043:
 1044: // Initialize with external Eikonal
 1045: void MGraniitti::Initialize(const MEikonal &eikonal_in) {
 1046:   // ** ALWAYS HERE ONLY AS LAST! **
 1047:   proc->post_Constructor();
 1048:
 1049:   // Set input eikonal
 1050:   proc->SetEikonal(eikonal_in);
 1051:
 1052:   // Integrate
 1053:   CallIntegrator(0);
 1054: }
 1055:
 1056: void MGraniitti::CallIntegrator(unsigned int N) {
 1057:   // Initialize global clock
 1058:   if (N == 0) { global_tictoc = MTimer(true); }
 1059:
 1060:   // Sample the phase space
 1061:   if (INTEGRATOR == "VEGAS") {
 1062:     SampleVegas(N);
 1063:   } else if (INTEGRATOR == "FLAT") {
 1064:     SampleFlat(N);
 1065:   } else if (INTEGRATOR == "NEURO") {
 1066:     SampleNeuro(N);
 1067:   } else {
 1068:     throw std::invalid_argument("MGraniitti::CallIntegrator: Unknown INTEGRATOR = " + INTEGRATOR +
 1069:                                 " (use VEGAS, FLAT, NEURO)");
 1070:   }
 1071: }
 1072:
 1073: // Initialize and generate events using VEGAS MC
 1074: void MGraniitti::SampleVegas(unsigned int N) {
 1075:   if (N == 0) {
 1076:     InitMultiMemory();
 1077:     GMODE = 0;  // Pure integration
 1078:   }
 1079:   if (N > 0) {
```



```
 1080:     GMODE = 1;  // Event generation
 1081:   }
 1082:
 1083:   // ************************************************
 1084:   // Create integral sampling region array
 1085:
 1086:   VD.InitRegion(proc->GetdLIPSDim());
 1087:
 1088:   // ************************************************
 1089:
 1090:   // Pure integration mode
 1091:   if (GMODE == 0) {
 1092:     const double MINTIME     = 0.1;  // Seconds
 1093:     unsigned int BURNIN_ITER = 3;    // BURN-IN iterations (default)!
 1094:
 1095:     // Initialize GRID
 1096:     do {
 1097:       unsigned int init   = 0;
 1098:       int          factor = 0;
 1099:
 1100:       do {  // Loop until stable
 1101:         factor       = VEGAS(init, vparam.NCALL, BURNIN_ITER, N);
 1102:         vparam.NCALL = vparam.NCALL * factor;
 1103:         if (factor == 1) { break; }  // We are done
 1104:       } while (true);
 1105:
 1106:       // Increase CALLS and re-run, if too fast
 1107:       if (itertime < MINTIME) {
 1108:         const double time_per_iter = itertime / vparam.NCALL;
 1109:
 1110:         // Max, because this is only minimum condition
 1111:         vparam.NCALL =
 1112:             std::max((unsigned int)vparam.NCALL, (unsigned int)(MINTIME / time_per_iter));
 1113:         VEGAS(init, vparam.NCALL, BURNIN_ITER, N);
 1114:       }
 1115:
 1116:       // Now re-calculate by skipping burn-in (init = 0) iterations
 1117:       // because they detoriate the integral value by bad grid
 1118:       init = 1;
 1119:       factor = VEGAS(init, vparam.NCALL, vparam.ITER, N);
 1120:
 1121:       if (factor == 1) {
 1122:         break;
 1123:       } else {
 1124:         BURNIN_ITER = 2 * BURNIN_ITER;
 1125:       }
 1126:     } while (true);
 1127:   }
 1128:
 1129:   // Event generation mode
 1130:   if (GMODE == 1) {
 1131:     const unsigned int init    = 2;
 1132:     const unsigned int itermin = 1E9;
 1133:     VEGAS(init, vparam.NCALL * 10, itermin, N);
 1134:   }
 1135: }
 1136:
 1137:
 1138: // Multithreaded VEGAS integrator
 1139: // [close to optimal importance sampling iff
 1140: //  integrand factorizes dimension by dimension]
 1141: //
 1142: // Original algorithm from:
 1143: // [REFERENCE: Lepage, G.P. Journal of Computational Physics, 1978]
 1144: // en.wikipedia.org/wiki/VEGAS_algorithm
 1145:
 1146: int MGraniitti::VEGAS(unsigned int init, unsigned int calls, unsigned int itermin, unsigned int N) {
 1147:   // First the initialization
 1148:   VD.Init(init, vparam);
 1149:
 1150:   if (init == 0 && !HILJAA) {
 1151:     gra::aux::ClearProgress();
 1152:     std::cout << rang::style::bold;
 1153:     printf("VEGAS burn-in iterations: \n\n");
 1154:     std::cout << rang::style::reset;
 1155:   }
 1156:
 1157:   if (init == 1 && !HILJAA) {
 1158:     gra::aux::ClearProgress();  // Progressbar clear
 1159:     gra::aux::PrintBar("-");
 1160:     std::cout << rang::style::bold;
 1161:     std::cout << "VEGAS importance sampling:" << std::endl << std::endl;
 1162:     std::cout << rang::style::reset;
```



```
 1163:     printf("<Multithreading CORES = %d> \n\n", CORES);
 1164:
 1165:     printf("~ dimension = %u \n", VD.FDIM);
 1166:     printf(" \n");
 1167:     printf("~ itermin   = %d \n", itermin);
 1168:     printf("~ calls     = %d \n", calls);
 1169:     printf("~ BINS      = %u \n", vparam.BINS);
 1170:     printf("~ PRECISION = %0.4f \n", vparam.PRECISION);
 1171:     printf("~ CHI2MAX   = %0.4f \n", vparam.CHI2MAX);
 1172:     printf("~ LAMBDA    = %0.4f \n", vparam.LAMBDA);
 1173:     printf("~ DEBUG     = %d \n", vparam.DEBUG);
 1174:     gra::aux::PrintBar("~");
 1175:   }
 1176:
 1177:   // Reset local timers
 1178:   local_tictoc = MTimer(true);
 1179:
 1180:   MTimer stime = MTimer(true);  // For statusprint
 1181:   MTimer atime = MTimer(true);  // For progressbar
 1182:
 1183:   // ------------------------------------------------------------
 1184:   // Create number of calls per thread, their sum == calls
 1185:   std::vector<unsigned int> LOCALcalls = VD.GetLocalCalls(calls, CORES);
 1186:
 1187:   MTimer gridtic;
 1188:
 1189:   // VEGAS grid iterations
 1190:   for (std::size_t iter = 0; iter < itermin; ++iter) {
 1191:     if (init == 0 && iter == 2) {          // Save time for one iteration
 1192:       itertime = gridtic.ElapsedSec() / 3;  // Average over 3 iter
 1193:     }
 1194:
 1195:     for (std::size_t j = 0; j < VD.FDIM; ++j) {
 1196:       for (std::size_t i = 0; i < vparam.BINS; ++i) {
 1197:         VD.fmat[i][j]  = 0.0;
 1198:         VD.f2mat[i][j] = 0.0;
 1199:       }
 1200:     }
 1201:     // Init here outside parallel processing
 1202:     VD.fsum  = 0.0;
 1203:     VD.f2sum = 0.0;
 1204:
 1205:     // ------------------------------------------------------------
 1206:     // SPAWN PARALLEL PROCESSING HERE
 1207:
 1208:     std::vector<std::thread> threads;
 1209:     for (int tid = 0; tid < CORES; ++tid) {
 1210:       threads.push_back(std::thread([=] { VEGASMultiThread(N, tid, init, LOCALcalls[tid]); }));
 1211:     }
 1212:
 1213:     // Loop again to join the threads
 1214:     for (auto &t : threads) { t.join(); }
 1215:     if (gra::globalExceptionPtr) {  // Exception handling of threads
 1216:       std::rethrow_exception(gra::globalExceptionPtr);
 1217:     }
 1218:
 1219:     // ------------------------------------------------------------
 1220:     // Estimates based on this iteration
 1221:
 1222:     VD.fsum *= 1.0 / static_cast<double>(calls);
 1223:     VD.f2sum *= pow2(1.0 / static_cast<double>(calls));
 1224:
 1225:     // Local integral estimate
 1226:     const double I_this = VD.fsum;
 1227:
 1228:     VD.f2sum = msqrt(VD.f2sum * calls);
 1229:     VD.f2sum = (VD.f2sum - VD.fsum) * (VD.f2sum + VD.fsum);  // - +
 1230:     if (VD.f2sum <= 0.0) { VD.f2sum = vparam.EPS; }
 1231:
 1232:     // Local error squared estimate
 1233:     const double I2_this = VD.f2sum * 1.0 / static_cast<double>(calls);
 1234:
 1235:     // ------------------------------------------------------------
 1236:     // Update global estimates
 1237:     const double alpha = 1.0 / I2_this;
 1238:
 1239:     VD.sumdata += alpha * I_this;
 1240:     VD.sumchi2 += alpha * pow2(I_this);
 1241:     VD.sweight += alpha;
 1242:
 1243:     // Integral and its error estimate
 1244:     stat.sigma     = VD.sumdata / VD.sweight;
 1245:     stat.sigma_err  = msqrt(1.0 / VD.sweight);
```



```
 1246:     stat.sigma_err2 = pow2(stat.sigma_err);
 1247:
 1248:     // Chi2
 1249:     double chi2this = (VD.sumchi2 - pow2(VD.sumdata) / VD.sweight) / (iter + 1E-5);
 1250:     chi2this        = (chi2this < 0) ? 0.0 : chi2this;
 1251:     stat.chi2       = chi2this;
 1252:     // ------------------------------------------------------------
 1253:
 1254:     // Got enough events generated
 1255:     if (GMODE == 1 && stat.generated >= N) { goto stop; }
 1256:
 1257:     // Fatal error and convergence restart treatment
 1258:     if (GMODE == 0 && iter > 0) {
 1259:       if (std::isnan(stat.sigma) || std::isinf(stat.sigma)) {
 1260:         gra::aux::ClearProgress();
 1261:         throw std::invalid_argument(
 1262:             "VEGAS:: Integral inf/nan: FATAL ERROR, cuts or parameters "
 1263:             "need fixing. Try <F> phase space class if not in use.");
 1264:       }
 1265:       if (stat.sigma < 1e-60) {
 1266:         gra::aux::ClearProgress();
 1267:         throw std::invalid_argument(
 1268:             "VEGAS:: Integral < 1E-60: Check VEGAS parameters, decaymode "
 1269:             "sanity, generation and fiducial cuts! Try <F> phase space class if not in use.");
 1270:       }
 1271:       if (stat.chi2 > 50) {
 1272:         gra::aux::ClearProgress();
 1273:         printf("VEGAS:: chi2 = %0.1f > 50: Convergence problem, increasing 10 x calls. \n",
 1274:                stat.chi2);
 1275:         printf("Try <F> phase space class if not in use. \n");
 1276:         return 10;  // 10 times more calls
 1277:       }
 1278:     }
 1279:
 1280:     if (vparam.DEBUG >= 0) {
 1281:       gra::aux::ClearProgress();
 1282:       printf(
 1283:           "VEGAS:: local iter = %4lu integral = %0.5E +- std = "
 1284:           "%0.5E \t [global integral = %0.5E +- std = %0.5E] \t "
 1285:           "chi2this = %0.2f \n",
 1286:           iter + 1, I_this, gra::math::msqrt(I2_this), stat.sigma, stat.sigma_err, chi2this);
 1287:
 1288:       if (vparam.DEBUG > 0) {
 1289:         for (std::size_t j = 0; j < VD.FDIM; ++j) {
 1290:           printf("VEGAS:: data for dimension j = %lu (VD.FDIM = %u) \n", j, VD.FDIM);
 1291:
 1292:           for (std::size_t i = 0; i < vparam.BINS; ++i) {
 1293:             printf(
 1294:                 "xmat[%3lu][j] = %0.5E, fmat[%3lu][j] = %0.5E "
 1295:                 "\n",
 1296:                 i, VD.xmat[i][j], i, VD.fmat[i][j]);
 1297:           }
 1298:         }
 1299:       }
 1300:     }
 1301:
 1302:     // We are not below the chi2 or precision condition -> increase iterations
 1303:     if (init > 0) {  // do not consider in burn-in (init = 0) mode
 1304:       if ((iter == (itermin - 1) && stat.chi2 > vparam.CHI2MAX) ||
 1305:           (iter == (itermin - 1) && stat.sigma_err / stat.sigma > vparam.PRECISION)) {
 1306:         ++itermin;
 1307:       }
 1308:     }
 1309:
 1310:     // Status and grid optimization
 1311:     if (GMODE == 0) {
 1312:       if (stime.ElapsedSec() > 2.0 || init == 0) {
 1313:         PrintStatus(stat.evaluations, N, local_tictoc, -1.0);
 1314:         stime.Reset();
 1315:       }
 1316:       if (atime.ElapsedSec() > 0.01) {
 1317:         gra::aux::PrintProgress((iter + 1) / static_cast<double>(itermin));
 1318:         atime.Reset();
 1319:       }
 1320:
 1321:       // ============================================================
 1322:       // **** Update VEGAS grid (not during event generation) ****
 1323:       VD.OptimizeGrid(vparam);
 1324:       // ============================================================
 1325:     }
 1326:
 1327:     // Unify histogram boundaries after burn-in across different threads
 1328:     // (due to adaptive histogramming)
```

```
1329:     if (init == 0 && iter == itermin - 1) { UnifyHistogramBounds(); }
1330:   }  // Main grid iteration loop
1331:
1332: stop:  // We jump here once finished (GOTO point)
1333:
1334:   // Final statistics
1335:   if (init > 0) { PrintStatistics(N); }
1336:
1337:   return 1;  // Return 1 for good
1338: }
1339:
1340: // This is called once for every VEGAS grid iteration
1341: void MGraniitti::VEGASMultiThread(unsigned int N, unsigned int THREAD_ID, unsigned int init,
1342:                                   unsigned int LOCALcalls) {
1343:   double zn = 0.0;
1344:   double zo = 0.0;
1345:   double ac = 0.0;
1346:
1347:   try {
1348:     for (std::size_t k = 0; k < LOCALcalls; ++k) {  // ** LOCALcalls ~= calls / CORES **
1349:
1350:       double vegasweight = 1.0;
1351:
1352:       // Phase space point vector
1353:       std::vector<double> xpoint(pvec[THREAD_ID]->GetdLIPSDim(), 0.0);
1354:
1355:       // Loop over dimensions and construct random vector xpoint
1356:       std::vector<double> indvec(VD.FDIM, 0);
1357:       for (std::size_t j = 0; j < VD.FDIM; ++j) {
1358:         // Draw random number
1359:         zn = pvec[THREAD_ID]->random.U(0, 1) * vparam.BINS + 1.0;
1360:         indvec[j] =
1361:             std::max((unsigned int)1, std::min((unsigned int)zn, (unsigned int)vparam.BINS));
1362:
1363:         if (indvec[j] > 1) {
1364:           zo = VD.xmat[indvec[j] - 1][j] - VD.xmat[indvec[j] - 2][j];
1365:           ac = VD.xmat[indvec[j] - 2][j] * (zn - indvec[j]) * zo;
1366:         } else {
1367:           zo = VD.xmat[indvec[j] - 1][j];
1368:           ac = (zn - indvec[j]) * zo;
1369:         }
1370:
1371:         // Multidim space vector component
1372:         xpoint[j] = VD.region[j] + ac * VD.dxvec[j];
1373:         vegasweight *= zo * vparam.BINS;
1374:
1375:       }  // VD.FDIM loop
1376:
1377:       // ****************************************************************
1378:       // ****** Call the process under integration to get the weight *******
1379:
1380:       gra::AuxIntData aux;
1381:       aux.vegasweight  = vegasweight;
1382:       aux.burn_in_mode = (init == 0) ? true : false;
1383:
1384:       const double W = pvec[THREAD_ID]->EventWeight(xpoint, aux);
1385:
1386:       // ****************************************************************
1387:
1388:       // @@ Multithreading lock (FAST SECTION) @@
1389:       gra::g_mutex.lock();
1390:
1391:       // Increase statistics
1392:       stat.Accumulate(aux);
1393:
1394:       // *** Importance weighting ***
1395:       const double f  = W * vegasweight;
1396:       const double f2 = pow2(f);
1397:
1398:       VD.fsum += f;
1399:       VD.f2sum += f2;
1400:
1401:       // Loop over dimensions and add importance weighted results
1402:       for (std::size_t j = 0; j < VD.FDIM; ++j) {
1403:         VD.fmat[indvec[j] - 1][j] += f;
1404:         VD.f2mat[indvec[j] - 1][j] += f2;
1405:       }
1406:
1407:       // ------------------------------------------------------------
1408:       // Initialization (integration) mode
1409:       if (GMODE == 0) {
1410:         // Do not consider burn-in phase weights (unstable)
1411:         if (init != 0) {
```



```
1412:           // Maximum raw weight (for general information, not used
1413:           // here)
1414:           if (W > stat.maxW) { stat.maxW = W; }
1415:           // Maximum total VEGAS importance weighted
1416:           if (f > stat.maxf) { stat.maxf = f; }
1417:         }
1418:       }
1419:       gra::g_mutex.unlock();
1420:       // @@ Multithreading unlock (FAST SECTION) @@
1421:
1422:       // ------------------------------------------------------------
1423:       // Event generation mode
1424:       if (GMODE == 1) {
1425:         // Enough events
1426:         if (stat.generated == (unsigned int)GetNumberOfEvents()) { break; }
1427:
1428:         // Event trial
1429:         SaveEvent(pvec[THREAD_ID], f, stat.maxf, aux);
1430:
1431:         if (THREAD_ID == 0 && atime.ElapsedSec() > 0.5) {
1432:           PrintStatus(stat.generated, N, local_tictoc, 10.0);
1433:           gra::aux::PrintProgress(stat.generated / static_cast<double>(N));
1434:           atime.Reset();
1435:         }
1436:       }
1437:     }  // calls loop
1438:   } catch (...) {
1439:     // Set the global exception pointer if exception arises
1440:     // This is because of multithreading
1441:     gra::globalExceptionPtr = std::current_exception();
1442:   }
1443: }
1444:
1445: // Generate events using plain simple MC (for reference/DEBUG purposes)
1446: void MGraniitti::SamplePlat(unsigned int N) {
1447:   // Integration mode
1448:   if (N == 0) {
1449:     GMODE = 0;
1450:     proc->PrintInit(HILJAA);
1451:   }
1452:   // Event generation mode
1453:   if (N > 0) { GMODE = 1; }
1454:
1455:   // Get dimension of the phase space
1456:   const unsigned int  dim = proc->GetdLIPSDim();
1457:   std::vector<double> randvec(dim, 0.0);
1458:
1459:   // Reset local timer
1460:   local_tictoc = MTimer(true);
1461:
1462:   // Progressbar
1463:   atime = MTimer(true);
1464:
1465:   // Reservation (test)
1466:   // using namespace std::placeholders; // _1, _2, ... come from here
1467:   // std::function<double(const std::vector<double>&, int, double,
1468:   // std::vector<double>&,
1469:   // bool&)> fifth =
1470:   //  std::bind(&MQuasiElastic::EventWeight, &proc_Q, _1, _2, _3, _4, _5);
1471:
1472:   // Event loop
1473:   while (true) {
1474:     // ** Generate new random numbers **
1475:     for (const auto &i : indices(randvec)) { randvec[i] = proc->random.U(0, 1); }
1476:
1477:     // Generate event
1478:     // Used for in-out control of the process
1479:     gra::AuxIntData aux;
1480:     aux.vegasweight  = 1.0;
1481:     aux.burn_in_mode = false;
1482:     const double W   = proc->EventWeight(randvec, aux);
1483:
1484:     // Increase statistics
1485:     stat.Accumulate(aux);
1486:     stat.Wsum += W;
1487:     stat.W2sum += pow2(W);
1488:
1489:     // Update cross section estimate
1490:     stat.CalculateCrossSection();
1491:
1492:     // Initialization
1493:     if (GMODE == 0) {
1494:       stat.maxW = (W > stat.maxW) ? W : stat.maxW;  // Update maximum weight
```



```
1495:       PrintStatus(stat.evaluations, N, local_tictoc, 10.0);
1496:
1497:       if ((stat.sigma_err / stat.sigma) < mcparam.PRECISION &&
1498:           stat.evaluations > mcparam.MIN_EVENTS) {
1499:         break;
1500:       }
1501:
1502:       // Progressbar
1503:       if (atime.ElapsedSec() > 0.1) {
1504:         gra::aux::PrintProgress(stat.evaluations / static_cast<double>(mcparam.MIN_EVENTS));
1505:         atime.Reset();
1506:       }
1507:     }
1508:
1509:     // Event generation mode
1510:     if (GMODE == 1) {
1511:       SaveEvent(proc, W, stat.maxW, aux);
1512:       PrintStatus(stat.generated, N, local_tictoc, 10.0);
1513:       if (stat.generated >= N) { break; }
1514:
1515:       // Progressbar
1516:       if (atime.ElapsedSec() > 0.1) {
1517:         gra::aux::PrintProgress(stat.generated / static_cast<double>(N));
1518:         atime.Reset();
1519:       }
1520:     }
1521:   }
1522:   PrintStatus(stat.generated, N, local_tictoc, -1.0);
1523:   PrintStatistics(N);
1524: }
1525:
1526: // Neural net Monte Carlo (small scale PROTOTYPE)
1527: void MGraniitti::SampleNeuro(unsigned int N) {
1528:   // Integration mode
1529:   if (N == 0) {
1530:     GMODE = 0;
1531:     proc->PrintInit(HILJAA);
1532:   }
1533:   // Event generation mode
1534:   if (N > 0) { GMODE = 1; }
1535:
1536:   // Get dimension of the phase space
1537:   const unsigned int D = proc->GetdLIPSDim();
1538:
1539:   // Reset timers
1540:   local_tictoc = MTimer(true);
1541:   atime        = MTimer(true);
1542:
1543:   // Reservation (test)
1544:   // using namespace std::placeholders; // _1, _2, ... come from here
1545:   // std::function<double(const std::vector<double>&, int, double,
1546:   // std::vector<double>&,
1547:   // bool&)> fifth =
1548:   // std::bind(&MQuasiElastic::EventWeight, &proc_Q, _1, _2, _3, _4, _5);
1549:
1550:   if (N == 0) {
1551:     // ------------------------------------------------------------
1552:     // Bind the object and call it
1553:     gra::neurojac::procptr = proc;  // First set address
1554:     // ------------------------------------------------------------
1555:
1556:     gra::neurojac::MNeuroJacobian neurojac;
1557:     gra::neurojac::BATCHSIZE = 100;
1558:
1559:     // Set network layer dimensions [first, ..., output]
1560:     gra::neurojac::par.D = D;  // Integrand dimension
1561:
1562:     gra::neurojac::par.L.push_back(gra::neurojac::Layer(2, D));  // Input
1563:     gra::neurojac::par.L.push_back(gra::neurojac::Layer(2, 2));
1564:     gra::neurojac::par.L.push_back(gra::neurojac::Layer(2, 2));
1565:     gra::neurojac::par.L.push_back(gra::neurojac::Layer(D, 2));  // Output
1566:
1567:     neurojac.Optimize();
1568:   }
1569:
1570:   // ------------------------------------------------------------------
1571:   // Now to the event generation
1572:
1573:   // Lambda capture
1574:   std::vector<double> u(D);
1575:
1576:   auto NeuroSample = [&](gra::AuxIntData &aux) {
1577:
```



```
1578:     // Prior p(z) distribution sampling
1579:     VectorXdual z(u.size());
1580:     for (std::size_t i = 0; i < D; ++i) { z[i] = proc->random.G(0, 1); }
1581:     const double p = val(gra::neurojac::gaussprob(z, 0, 1));
1582:
1583:     // Evaluate network map
1584:     VectorXdual u_ = gra::neurojac::G_net(z);
1585:
1586:     // Evaluate the Jacobian matrix du/dz
1587:     MatrixXd dudz = jacobian(gra::neurojac::G_net, u_, z);
1588:
1589:     // Abs Jacobian determinant and inverse prior
1590:     const double jacweight = abs(dudz.determinant()) / p;
1591:
1592:     for (std::size_t i = 0; i < D; ++i) { u[i] = val(u_[i]); }
1593:
1594:     // Evaluate event weight
1595:     aux.vegasweight     = jacweight;
1596:     aux.burn_in_mode    = false;
1597:     const double weight = proc->EventWeight(u, aux) * jacweight;
1598:
1599:     return weight;
1600:   };
1601:
1602:   // Event loop
1603:   while (true) {
1604:     // aux used for in-out control of the process
1605:     gra::AuxIntData aux;
1606:     const double    W = NeuroSample(aux);
1607:
1608:     // Increase statistics
1609:     stat.Accumulate(aux);
1610:     stat.Wsum += W;
1611:     stat.W2sum += pow2(W);
1612:
1613:     // Update cross section estimate
1614:     stat.CalculateCrossSection();
1615:
1616:     // Initialization
1617:     if (GMODE == 0) {
1618:       stat.maxW = (W > stat.maxW) ? W : stat.maxW;  // Update maximum weight
1619:       PrintStatus(stat.evaluations, N, local_tictoc, 10.0);
1620:
1621:       if ((stat.sigma_err / stat.sigma) < mcparam.PRECISION &&
1622:           stat.evaluations > mcparam.MIN_EVENTS) {
1623:         break;
1624:       }
1625:
1626:       // Progressbar
1627:       if (atime.ElapsedSec() > 0.1) {
1628:         gra::aux::PrintProgress(stat.evaluations / static_cast<double>(mcparam.MIN_EVENTS));
1629:         atime.Reset();
1630:       }
1631:     }
1632:
1633:     // Event generation mode
1634:     if (GMODE == 1) {
1635:       SaveEvent(proc, W, stat.maxW, aux);
1636:       PrintStatus(stat.generated, N, local_tictoc, 10.0);
1637:       if (stat.generated >= N) { break; }
1638:
1639:       // Progressbar
1640:       if (atime.ElapsedSec() > 0.1) {
1641:         gra::aux::PrintProgress(stat.generated / static_cast<double>(N));
1642:         atime.Reset();
1643:       }
1644:     }
1645:   }
1646:   PrintStatus(stat.generated, N, local_tictoc, -1.0);
1647:   PrintStatistics(N);
1648: }
1649:
1650: // Save unweighted or weighted event
1651: int MGraniitti::SaveEvent(MProcess *pr, double weight, double MAXWEIGHT,
1652:                           const gra::AuxIntData &aux) {
1653:   gra::g_mutex.lock();
1654:   stat.trials += 1;  // This is one trial more
1655:
1656:   // 0. Hit-Miss
1657:   bool hit_in = proc->random.U(0, 1) < (weight / MAXWEIGHT);
1658:
1659:   // 2. We see weight larger than maxweight
1660:   if (!WEIGHTED && weight > MAXWEIGHT) {
```

```
1661:     ++stat.N_overflow;
1662:     hit_in = false;
1663:   }
1664:   gra::g_mutex.unlock();
1665:
1666:   // Three ways to accept the event. N.B. weight > 0 needed if the amplitude
1667:   // fails numerically
1668:   if (hit_in && aux.Valid()) || (WEIGHTED && aux.Valid()) || aux.forced_accept) {
1669:     // Create HepMC3 event (do not lock yet for speed)
1670:     HepMC3::GenEvent evt(HepMC3::Units::GEV, HepMC3::Units::MM);
1671:
1672:     // Construct event record
1673:     if (!pr->EventRecord(evt)) {  // Event not ok!
1674:       // std::cout << "MGraniitti::SaveEvent: Last moment rare veto!" <<
1675:       // std::endl;
1676:       return 2;
1677:     }
1678:
1679:     // Event ok, continue >>
1680:
1681:     // @@@ THIS IS THREAD-NON-SAFE -> LOCK IT
1682:     gra::g_mutex.lock();
1683:
1684:     // ** This is a multithreading race-condition treatment **
1685:     if (stat.generated == (unsigned int)GetNumberOfEvents()) {
1686:       stat.trials -= 1;  // Correct statistics, unnecessary trial
1687:       gra::g_mutex.unlock();
1688:       return 1;
1689:     }
1690:
1691:     // Save cross section information (HepMC3 format wants it event
1692:     // by event)
1693:     std::shared_ptr<HepMC3::GenCrossSection> xsobj = std::make_shared<HepMC3::GenCrossSection>();
1694:     evt.add_attribute("GenCrossSection", xsobj);
1695:
1696:     // Now add the value in picobarns [HepMC3 convention]
1697:     if (xsforced > 0) {
1698:       xsobj->set_cross_section(xsforced * 1E12, 0);  // external fixed one
1699:     } else {
1700:       xsobj->set_cross_section(stat.sigma * 1E12, stat.sigma_err * 1E12);
1701:     }
1702:
1703:     // Save event weight (unweighted events with weight 1)
1704:     const double HepMC3_weight = WEIGHTED ? weight : 1.0;
1705:     evt.weights().push_back(HepMC3_weight);  // add more weights with .push_back()
1706:
1707:     if (FORMAT == "hepmc3") {
1708:       outputHepMC3->write_event(evt);
1709:     } else if (FORMAT == "hepmc2") {
1710:       outputHepMC2->write_event(evt);
1711:     } else if (FORMAT == "hepevt") {
1712:       outputHEPEVT->write_event(evt);
1713:     } else {
1714:       throw std::invalid_argument("MGraniitti::SaveEvent: Unknown output FORMAT " + FORMAT);
1715:     }
1716:
1717:     // LAST STEP
1718:     stat.generated += 1;  // +1 event generated
1719:
1720:     gra::g_mutex.unlock();
1721:     // @@ THIS IS THREAD-NON-SAFE <- LOCK IT @@
1722:     return 0;
1723:   } else {
1724:     return 1;
1725:   }
1726: }
1727:
1728: // Intermediate statistics
1729: void MGraniitti::PrintStatus(unsigned int events, unsigned int N, MTimer &tictoc, double timercut) {
1730:   if (tictoc.ElapsedSec() > timercut) {
1731:     tictoc.Reset();
1732:     gra::aux::ClearProgress();
1733:
1734:     double       peak_use     = 0.0;
1735:     double       resident_use = 0.0;
1736:     const double MB           = 1024 * 1024;
1737:     const double GB           = MB * 1024;
1738:     gra::aux::GetProcessMemory(peak_use, resident_use);
1739:     peak_use /= MB;
1740:     resident_use /= MB;
1741:
1742:     if (GMODE == 0) {
1743:       const double global_lap = global_tictoc.ElapsedSec();
```



```
1744:       printf(
1745:           "[%0.1f MB] xs: %9.3E, er: %7.5f [chi2 = %2.1f], %4.1f "
1746:           "min ~ %0.1E Hz \n",
1747:           resident_use, stat.sigma, stat.sigma_err / stat.sigma, stat.chi2, global_lap / 60.0,
1748:           events / global_lap);
1749:     }
1750:     if (GMODE == 1) {
1751:       const double global_lap     = global_tictoc.ElapsedSec() - time_t0;
1752:       double       outputfilesize = gra::aux::GetFileSize(FULL_OUTPUT_STR) / GB;
1753:
1754:       printf(
1755:           "[%0.1f MB/%0.2f GB] E: %9d, xs: %9.3E, er: %7.5f, %0.1f/%0.1f min ~ "
1756:           "%0.1E Hz \n",
1757:           resident_use, outputfilesize, events, stat.sigma, stat.sigma_err / stat.sigma,
1758:           global_lap / 60.0, (N - events) * global_lap / (double)events / 60.0,
1759:           events / global_lap);
1760:     }
1761:   }
1762: }
1763:
1764: // Final statistics
1765: void MGraniitti::PrintStatistics(unsigned int N) {
1766:   gra::aux::ClearProgress();  // Clear progressbar
1767:
1768:   if (GMODE == 0) {
1769:     time_t0 = global_tictoc.ElapsedSec();
1770:     gra::aux::PrintBar("=");
1771:     std::cout << rang::style::bold;
1772:     printf("Monte Carlo integration: \n\n\n");
1773:     std::cout << rang::style::reset;
1774:
1775:     if (proc->GetISOLATE()) {
1776:       std::cout << rang::fg::red << "NOTE: Central leg phase space isolation tag &> in use!";
1777:       std::cout << rang::fg::reset;
1778:       std::cout << std::endl << std::endl;
1779:     }
1780:
1781:     // Check if we have cascaded phase space turned on in the x-section calculation
1782:     double prod    = 1.0;
1783:     double prod2pi = 1.0;
1784:     double volume  = 1.0;
1785:     int    Nf      = 0;
1786:     for (const auto &i : indices(proc->lts.decaytree)) {
1787:       proc->CalculatePhaseSpace(proc->lts.decaytree[i], prod, prod2pi, volume, Nf);
1788:     }
1789:     unsigned int N_leg = std::max((int)proc->lts.decaytree.size(), Nf) + 2;  // +2 forward legs
1790:
1791:     // Special cases
1792:     if (proc->GetISOLATE()) { N_leg = 3; }        // ISOLATEd 2->3 process with <F> phase space
1793:     if (proc->GetdLIPSDim() == 2) { N_leg = 2; }  // <P> and <Q> class
1794:
1795:     printf("(2->%d cross section):              [%0.3E +- %0.3E] barn \n", N_leg, stat.sigma,
1796:            stat.sigma_err);
1797:     printf("Sampling uncertainty:              %0.3f %% \n", 100 * stat.sigma_err / stat.sigma);
1798:     std::cout << std::endl;
1799:
1800:     if (proc->lts.DW_sum.Integral() > 0) {  // Recursive phase space on
1801:
1802:       std::cout << std::endl;
1803:       const double PSvolume = proc->lts.DW_sum.Integral();
1804:       // const double PSvolume_error     = proc->lts.DW_sum.IntegralError();
1805:
1806:       const double PSvolume_exact       = proc->lts.DW_sum_exact.Integral();
1807:       const double PSvolume_exact_error = proc->lts.DW_sum_exact.IntegralError();
1808:
1809:       // Get sum of decay daughter masses
1810:       double MSUM = 0.0;
1811:       for (std::size_t i = 0; i < proc->lts.decaytree.size(); ++i) {
1812:         MSUM += proc->lts.decaytree[i].m_offshell;
1813:       }
1814:       // only 2-body exact or massless case
1815:       if (proc->lts.decaytree.size() == 2 || MSUM < 1e-6) {
1816:         printf("Analytic phase space volume:       %0.3E +- %0.3E \n", PSvolume_exact,
1817:                PSvolume_exact_error);
1818:         printf("RATIO: MC/analytic:                %0.6f \n", PSvolume / PSvolume_exact);
1819:       }
1820:       std::cout << std::endl;
1821:
1822:       // Print out phase space weight
1823:       printf("(1->%lu LIPS):                     %0.3E +- %0.3E  ", proc->lts.decaytree.size(),
1824:              proc->lts.DW_sum.Integral(), proc->lts.DW_sum.IntegralError());
1825:
1826:       if (proc->GetISOLATE()) {
```



```
1827:         std::cout << "[" << rang::fg::red << "INACTIVE " << rang::fg::reset
1828:                   << "part of integral  <=> apply manual BR]" << std::endl;
1829:       } else {
1830:         std::cout << "[" << rang::fg::green << "ACTIVE " << rang::fg::reset
1831:                   << "part of integral / (2PI)]" << std::endl;
1832:       }
1833:       // Recursion relation based (phase space factorization):
1834:       // d^N PS(s; p_1, p2, ...p_N)
1835:       // = 1/(2*PI) * d^3 PS(s; p1,p2,pX) d^(N-2) PS(M^2; pX,p3,p4,..,pN) dM^2
1836:       //
1837:       // Decaywidth = 1/(2M S) \int dPS |M_decay|^2, where M
1838:       // = mother mass, S = final state symmetry factor
1839:
1840:       if (!proc->GetISOLATE() && proc->GetdLIPSDim() != 2) {  // Special cases
1841:         printf("\n\n(2->3 cross section ~= [2->%u / (1->%lu LIPS) x 2PI]):      %0.3E \n", N_leg,
1842:                proc->lts.decaytree.size(), stat.sigma / proc->lts.DW_sum.Integral() * (2 * PI));
1843:         std::cout << std::endl;
1844:         printf("** Remember to use &> operator instead of -> for phase space isolation ** \n\n");
1845:       }
1846:     }
1847:
1848:     // Print recursively
1849:     double product    = 1.0;
1850:     double product2pi = 1.0;
1851:     int    N_final    = 2;  // forward legs == 2
1852:     for (const auto &i : indices(proc->lts.decaytree)) {
1853:       proc->PrintPhaseSpace(proc->lts.decaytree[i], product, product2pi, N_final);
1854:       if (proc->lts.decaytree[i].legs.size() != 0) { std::cout << std::endl; }
1855:     }
1856:     gra::aux::PrintBar("-");
1857:
1858:     std::cout << std::endl;
1859:     printf("Integration runtime:              %0.2f sec \n", time_t0);
1860:     printf("Integrand sampling frequency:     %0.2E Hz \n",
1861:            (stat.evaluations - stat.trials) / time_t0);
1862:     std::cout << std::endl;
1863:     printf("<Statistics> \n");
1864:     printf("MAX raw integrand weight:         %0.3E \n", stat.maxW);
1865:     if (INTEGRATOR == "VEGAS") { printf("MAX weight (vegas x integrand):   %0.3E \n", stat.maxf); }
1866:
1867:     printf("\n");
1868:     printf("Amplitude passing rate:           %0.3E \n", stat.amplitude_ok / stat.evaluations);
1869:     printf("Kinematics passing rate:          %0.3E \n", stat.kinematics_ok / stat.evaluations);
1870:     printf("Fiducial cuts passing rate:       %0.3E \n", stat.fidcuts_ok / stat.evaluations);
1871:     printf("Veto cuts passing rate:           %0.3E \n", stat.vetocuts_ok / stat.evaluations);
1872:     printf("\n");
1873:
1874:     std::cout << std::endl;
1875:     printf(
1876:         "** All values include phase space generation and "
1877:         "fiducial (+ veto) cuts ** \n");
1878:     gra::aux::PrintBar("=");
1879:   }
1880:   if (GMODE == 1) {
1881:     const double lap = global_tictoc.ElapsedSec() - time_t0;
1882:
1883:     gra::aux::PrintBar("=");
1884:     if (WEIGHTED) {
1885:       gra::aux::PrintNotice();
1886:       std::cout << rang::fg::red << "You did WEIGHTED event generation:" << std::endl
1887:                 << std::endl
1888:                 << std::endl;
1889:     } else {
1890:       std::cout << rang::fg::green
1891:                 << "You did UNWEIGHTED (acceptance-rejection) event generation:" << std::endl
1892:                 << std::endl
1893:                 << std::endl;
1894:     }
1895:     std::cout << rang::style::reset;
1896:
1897:     printf("Generation efficiency:    %0.3E (%d / %0.0f) \n", N / (double)stat.trials, N,
1898:            stat.trials);
1899:     printf("Weight overflow:          %0.3E (%d / %0.0f) \n", stat.N_overflow / (double)stat.trials,
1900:            stat.N_overflow, stat.trials);
1901:     printf("Generation runtime:       %0.2f sec \n", lap);
1902:     printf("Generation frequency:     %0.2E Hz \n", N / lap);
1903:
1904:     double outputfilesize = gra::aux::GetFileSize(FULL_OUTPUT_STR) / (1024.0 * 1024.0 * 1024.0);
1905:
1906:     printf("Outputfile size:          %0.3f GB [%s] \n", outputfilesize, FULL_OUTPUT_STR.c_str());
1907:     gra::aux::PrintBar("=");
1908:     std::cout << std::endl;
1909:   }
```



```
1910: }
1911:
1912: }  // namespace gra
1913:
```

```cpp
// "Brute force" Dirac Gamma Algebra for amplitudes
//
// Next step, optimize this class using more efficient spinor-products.
// For those, see e.g.:
//
// www.phys.nthu.edu.tw/~class/Helicity/YPYAO_Note_01.pdf
// www.physics.wayne.edu/~ablechman/main/Research_files/shm.pdf
// R. Kleiss, W.J. Stirling, Spinor Techniques, Nucl.Phys. B262 (1985) 235-262
// P. Richarsson, arxiv.org/abs/hep-ph/0110108
//
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

// C++
#include <complex>
#include <ostream>
#include <vector>

// Tensor algebra
#include "FTensor.hpp"

// Own
#include "Granitti/M4Vec.h"
#include "Granitti/MAux.h"
#include "Granitti/MDirac.h"
#include "Granitti/MKinematics.h"
#include "Granitti/MMath.h"
#include "Granitti/MSpin.h"

using FTensor::Tensor1;
using FTensor::Tensor2;

using gra::aux::indices;
using gra::math::msqrt;
using gra::math::pow2;
using gra::math::zi;

namespace gra {
MDirac::MDirac() { InitGammaMatrices("DIRAC"); }

MDirac::MDirac(const std::string &basis) { InitGammaMatrices(basis); }

void MDirac::InitGammaMatrices(const std::string &basis) {
  // Set gamma basis
  if (basis == "DIRAC") {
    BASIS = "D";
  } else if (basis == "CHIRAL") {
    BASIS = "C";
  } else {
    throw std::invalid_argument("MDirac::InitGammaMatrices: Unknown gamma basis chosen: " + basis);
  }

  // PARITY OPERATOR = y^0
  // Intrinsic parity of fermions +1, anti-fermion -1

  const MMatrix<std::complex<double>> y0_chiral{
      std::vector<std::complex<double>>{0.0, 0.0, 1.0, 0.0},
      std::vector<std::complex<double>>{0.0, 0.0, 0.0, 1.0},
      std::vector<std::complex<double>>{1.0, 0.0, 0.0, 0.0},
      std::vector<std::complex<double>>{0.0, 1.0, 0.0, 0.0}};

  const MMatrix<std::complex<double>> y0_dirac{
      std::vector<std::complex<double>>{1.0, 0.0, 0.0, 0.0},
      std::vector<std::complex<double>>{0.0, 1.0, 0.0, 0.0},
      std::vector<std::complex<double>>{0.0, 0.0, -1.0, 0.0},
      std::vector<std::complex<double>>{0.0, 0.0, 0.0, -1.0}};

  // \equiv i \gamma^0\gamma^1\gamma^2\gamma^3
  const MMatrix<std::complex<double>> y5_chiral{
      std::vector<std::complex<double>>{-1.0, 0.0, 0.0, 0.0},
      std::vector<std::complex<double>>{0.0, -1.0, 0.0, 0.0},
      std::vector<std::complex<double>>{0.0, 0.0, 1.0, 0.0},
      std::vector<std::complex<double>>{0.0, 0.0, 0.0, 1.0}};

  // \equiv i \gamma^0\gamma^1\gamma^2\gamma^3
  const MMatrix<std::complex<double>> y5_dirac{
      std::vector<std::complex<double>>{0.0, 0.0, 1.0, 0.0},
      std::vector<std::complex<double>>{0.0, 0.0, 0.0, 1.0},
      std::vector<std::complex<double>>{1.0, 0.0, 0.0, 0.0},
      std::vector<std::complex<double>>{0.0, 1.0, 0.0, 0.0}};

  // ---------------------------------------------------------------
```



```cpp
  // Both chiral and dirac basis
  // Contravariant matrices (upper index)

  MMatrix<std::complex<double>> y1_up{std::vector<std::complex<double>>{0.0, 0.0, 0.0, 1.0},
                                      std::vector<std::complex<double>>{0.0, 0.0, 1.0, 0.0},
                                      std::vector<std::complex<double>>{0.0, -1.0, 0.0, 0.0},
                                      std::vector<std::complex<double>>{-1.0, 0.0, 0.0, 0.0}};

  MMatrix<std::complex<double>> y2_up{std::vector<std::complex<double>>{0.0, 0.0, 0.0, -zi},
                                      std::vector<std::complex<double>>{0.0, 0.0, zi, 0.0},
                                      std::vector<std::complex<double>>{0.0, zi, 0.0, 0.0},
                                      std::vector<std::complex<double>>{-zi, 0.0, 0.0, 0.0}};

  MMatrix<std::complex<double>> y3_up{std::vector<std::complex<double>>{0.0, 0.0, 1.0, 0.0},
                                      std::vector<std::complex<double>>{0.0, 0.0, 0.0, -1.0},
                                      std::vector<std::complex<double>>{-1.0, 0.0, 0.0, 0.0},
                                      std::vector<std::complex<double>>{0.0, 1.0, 0.0, 0.0}};

  // ---------------------------------------------------------------
  // Contravariant and covariant
  if (BASIS == "D") {
    gamma_up = {y0_dirac, y1_up, y2_up, y3_up, y5_dirac};
    gamma_lo = {y0_dirac, -y1_up, -y2_up, -y3_up, y5_dirac};
  }
  if (BASIS == "C") {
    gamma_up = {y0_chiral, y1_up, y2_up, y3_up, y5_chiral};
    gamma_lo = {y0_chiral, -y1_up, -y2_up, -y3_up, y5_chiral};
  }

  // sigma = i/2 [\gamma^\mu, \gamma^\nu]
  // \sigma_{\mu\nu} each [mu][nu] contains one MMatrix
  sigma_up = std::vector<std::vector<MMatrix<std::complex<double>>>>(
      4, std::vector<MMatrix<std::complex<double>>>(4, MMatrix<std::complex<double>>(4, 4)));
  sigma_lo = std::vector<std::vector<MMatrix<std::complex<double>>>>(
      4, std::vector<MMatrix<std::complex<double>>>(4, MMatrix<std::complex<double>>(4, 4)));

  for (const auto &mu : LI) {
    for (const auto &nu : LI) {
      sigma_up[mu][nu] = (gamma_up[mu] * gamma_up[nu] - gamma_up[nu] * gamma_up[mu]) * (zi / 2.0);
      sigma_lo[mu][nu] = (gamma_lo[mu] * gamma_lo[nu] - gamma_lo[nu] * gamma_lo[mu]) * (zi / 2.0);
    }
  }
}

// Chirality projectors, valid for all gamma representations
// Right handed
MMatrix<std::complex<double>> MDirac::PR() const { return (I4 + gamma_up[4]) * 0.5; }
// Left handed
MMatrix<std::complex<double>> MDirac::PL() const { return (I4 - gamma_up[4]) * 0.5; }

// Two component Weyl (chiral) spinor
//
std::vector<std::complex<double>> MDirac::XiSpinor(const M4Vec &p, int helicity) const {
  const double theta2 = p.Theta() / 2.0;
  const double phi    = p.Phi();
  const double c      = std::cos(theta2);
  const double s      = std::sin(theta2);

  if (helicity == 1) { return {c, s * std::exp(zi * phi)}; }
  if (helicity == -1) { return {-s * std::exp(-zi * phi), c}; }
  throw std::invalid_argument("MDirac::XiSpinor: helicity is not -1 or 1");
}

// ---------------------------------------------------------------
// Helicity eigenstate spinors for particles
//
// <@@ DEFINED IN CHIRAL GAMMA MATRIX REPRESENTATION @@>
//
std::vector<std::complex<double>> MDirac::uHelChiral(const M4Vec &p, int helicity) const {
  if (BASIS != "C") { throw std::invalid_argument("MDirac::uHelChiral: Wrong gamma basis in use"); }

  const double E      = p.E();
  const double p3     = p.P3mod();
  const double m      = p.M();
  const double theta2 = p.Theta() / 2.0;
  const double phi    = p.Phi();

  const double              c     = std::cos(theta2);
  const double              s     = std::sin(theta2);
  const std::complex<double> phase = std::exp(zi * phi);

  const double neg = E + m - p3;
```



```cpp
  const double pos = E + m + p3;
  const double N   = 1.0 / (std::sqrt(2.0) * msqrt(E + m));  // Volume normalization to 2E

  switch (helicity) {
    case 1:
      return {c * neg * N, s * phase * neg * N, c * pos * N, s * phase * pos * N};
    case -1:
      return {-s * pos * N, c * phase * pos * N, -s * neg * N, c * phase * neg * N};
  }

  throw std::invalid_argument("MDirac::uHelChiral: helicity is not -1 or 1");
}

// Helicity eigenstate spinors for Anti-Particles
//
// <@@ DEFINED IN CHIRAL GAMMA MATRIX REPRESENTATION @@>
//
std::vector<std::complex<double>> MDirac::vHelChiral(const M4Vec &p, int helicity) const {
  if (BASIS != "C") { throw std::invalid_argument("MDirac::vHelChiral: Wrong gamma basis in use"); }

  if (helicity != 1 && helicity != -1) {
    throw std::invalid_argument("MDirac::vHelChiral: helicity is not -1 or 1");
  }
  // Flip the helicity, so we can use particle solution permutated
  helicity                          = -helicity;
  const std::vector<std::complex<double>> v = uHelChiral(p, helicity);
  return {-v[0], -v[1], v[2], v[3]};

  throw std::invalid_argument("MDirac::vHelChiral: helicity is not -1 or 1");
}
// ---------------------------------------------------------------------------

// ---------------------------------------------------------------------------
// Helicity eigenstate spinors
//
// <@@ DEFINED IN DIRAC GAMMA MATRIX REPRESENTATION @@>
//
// [REFERENCE: Thomson, Modern Particle Physics, Cambridge University Press]
// [https://www.hep.phy.cam.ac.uk/~thomson/partIIIparticles/handouts/Handout_2_2011.pdf]
//
// In high energy limit E >> m, helicity eigenstates are eigenstates of gamma^5
// gamma^5 u_+ = +u_+
// gamma^5 u_- = -u_-
// gamma^5 v_+ = -v_+
// gamma^5 v_- = +v_-
//
// Remember: In the limit E >> m (only then)
// -> left and right handed chiral states == helicity states.
//
std::vector<std::complex<double>> MDirac::uHelDirac(const M4Vec &p, int helicity) const {
  if (BASIS != "D") { throw std::invalid_argument("MDirac::uHelDirac: Wrong gamma basis in use"); }

  const double E  = p.E();
  const double p3 = p.P3mod();
  const double m  = p.M();
  const double N  = msqrt(E + m);  // Volume normalization to 2E

  const double theta2 = p.Theta() / 2.0;
  const double phi    = p.Phi();

  const double              c     = std::cos(theta2);
  const double              s     = std::sin(theta2);
  const std::complex<double> phase = std::exp(zi * phi);

  switch (helicity) {
    case 1:
      return {N * c, N * phase * s, N * p3 / (E + m) * c, N * p3 / (E + m) * phase * s};
    case -1:
      return {-N * s, N * phase * c, N * p3 / (E + m) * s, -N * p3 / (E + m) * phase * c};
  }

  throw std::invalid_argument("MDirac::uHelDirac: helicity is not -1 or 1");
}

// Helicity eigenstate spinors for Anti-Fermions
//
// <@@ DEFINED IN DIRAC GAMMA-MATRIX REPRESENTATION @@>
//
std::vector<std::complex<double>> MDirac::vHelDirac(const M4Vec &p, int helicity) const {
  if (BASIS != "D") { throw std::invalid_argument("MDirac::vHelDirac: Wrong gamma basis in use"); }
  if (helicity != 1 && helicity != -1) {
    throw std::invalid_argument("MDirac::vHelDirac: helicity is not -1 or 1");
  }
```



```cpp
  // Flip the helicity, so we can use particle solution permutated
  helicity                          = -helicity;
  const std::vector<std::complex<double>> v = uHelDirac(p, helicity);
  return {v[2], v[3], v[0], v[1]};
}
// ---------------------------------------------------------------------------

// ---------------------------------------------------------------------------
// Dirac-Spinor "spin eigenstate" fermions: (\gamma^\mu p_\mu - m)u = 0
//
// <@@ DEFINED IN DIRAC GAMMA MATRIX REPRESENTATION @@>
//
std::vector<std::complex<double>> MDirac::uDirac(const M4Vec &p, int spin) const {
  if (BASIS != "D") { throw std::invalid_argument("MDirac::uDirac: Wrong gamma basis in use"); }
  if (spin != -1 && spin != 1) {
    throw std::invalid_argument("MDirac::uDirac: spin state argument != -1 or 1");
  }
  const double E = p.E();
  const double m = p.M();
  const double N = msqrt(E + m);  // Normalization to 2E per unit volume

  if (spin == 1) {
    return {N * 1.0, 0.0, N * p[3] / (E + m), N * (p[1] + zi * p[2]) / (E + m)};
  } else {  // spin == -1
    return {0.0, N * 1.0, N * (p[1] - zi * p[2]) / (E + m), N * (-p[3] / (E + m))};
  }
}

// Dirac-Spinor "spin eigenstate" for anti-fermions: (\gamma^\mu p_\mu + m)v = 0
//
// <@@ DEFINED IN DIRAC GAMMA MATRIX REPRESENTATION @@>
//
std::vector<std::complex<double>> MDirac::vDirac(const M4Vec &p, int spin) const {
  if (BASIS != "D") { throw std::invalid_argument("MDirac::vDirac: Wrong gamma basis in use"); }
  if (!(spin == -1 || spin == 1)) {
    throw std::invalid_argument("MDirac::vDirac: spin state argument != -1 or 1");
  }
  // Flip the spin, then use the u-particle solution permutated
  spin                                = -spin;
  const std::vector<std::complex<double>> v = uDirac(p, spin);
  return {v[2], v[3], v[0], v[1]};
}
// ---------------------------------------------------------------------------

// Return Hermitician angular momentum operators J_i = 1/2 \sigma_i, i = 1,2,3
MMatrix<std::complex<double>> MDirac::J_operator(unsigned int i) const {
  if (i == 1) { return sigma_x * 0.5; }
  if (i == 2) { return sigma_y * 0.5; }
  if (i == 3) { return sigma_z * 0.5; }

  throw std::invalid_argument("MDirac::J_operator: i is invalid (not 1,2,3)");
}

// Transformation bilinears:
//
// Vector current: \bar{u}_1 \gamma^\mu u_2
// Axial current:  \bar{u}_1 \gamma^\mu \gamma_5 \u_2
// Tensor current: \bar{u}_1 \sigma^{\mu\nu} \gamma_5 \u_2
// Pseudoscalar current:  \bar{u}_1 \gamma_5 \u_2
//
//
// Adjoing gamma matrix: gamma^\mu\dagger = gamma^0 gamma^\mu gamma^0
//
//
// Charge conjugation operator matrix is basis dependent.
// u = C \bar{v}^T <=> \bar{v}^T = C^{-1} u
//
//
// Basis independent definition:
// C^{-1} \gamma_\mu C = -\gamma_\mu^T   <=>  C \gamma^\mu C^{-1} = -\gamma^\mu
// C^\dagger = C^{-1}
// C^T = -C
//
MMatrix<std::complex<double>> MDirac::C_up() const {
  if (BASIS == "D" || BASIS == "C") {
    return -gamma_up[2] * gamma_up[0] * zi;
  } else {
    throw std::invalid_argument("MDirac::C0P: Unknown gamma basis");
  }
}
}
```

```
333: // ------------------------------------------------------------------------
334: // Basic notes:
335: //
336: // 1. Helicity is conserved in fermion lines
337: // 2. By gauge invariance, polarization vectors can be chosen with explicit
338: //    representation. Invidual sub-amplitudes are gauge dependent, their sum is
339: //    not.
340: // 3. Helicity amplitudes do not interfere, sum incoherently \sum_{config}
341: // |M_i|^2
342: //
343: //
344: // ------------------------------------------------------------------------
345: // Conventions:
346: //
347: // Spin-1/2:
348: // incoming particle:      u(p)
349: // outgoing particle:      \bar{u}(p)
350: // incoming anti-particle: \bar{v}(p)
351: // outgoing anti-particle: v(p)
352: //
353: // Propagators:
354: // spin 1:                 -ig_\mu\nu / q^2
355: // spin 1/2:                i(\gamma^\mu q_\mu + m)/(q^2 - m^2)
356: //
357: // Vertex factor:
358: //                         ie\gamma^\mu
359: //
360: // Matrix element: -iM     product of rules
361: // ------------------------------------------------------------------------
362: //
363: // Photon propagator: -ig_{\mu\nu} / q^2
364: //
365: // Remember:
366: // \sum_{4 virtual polarizations} \eps_\mu^\lambda (\eps_{\nu}^\lambda)^* =
367: // -g_{\mu\nu}
368: //
369: Tensor2<std::complex<double>, 4, 4> MDirac::iD_y(const double q2) const {
370:   Tensor2<std::complex<double>, 4, 4> T;
371:
372:   for (const auto &u : LI) {
373:     for (const auto &v : LI) { T(u, v) = -zi * g[u][v] / q2; }
374:   }
375:   return T;
376: }
377:
378: // Internal Fermion propagator (matrix with spinor indices, no Lorentz indices)
379: //
380: // Input as contravariant (upper) index 4-vector
381: //
382: MMatrix<std::complex<double>> MDirac::iD_F(const M4Vec &q, double m) const {
383:   return (FSlash(q) + I4 * m) * (zi / (q.M2() - pow2(m)));
384: }
385:
386: // Polarization vector eps^{m}_\mu(k) for massless spin-1 with helicity m = -1,1
387: //
388: // Spatial dependence only on the direction.
389: //
390: // [REFERENCE: http://scipp.ucsc.edu/~haber/ph218/polsum.pdf]
391: // {t,x,y,z} order convention!
392: //
393: Tensor1<std::complex<double>, 4> MDirac::EpsSpin1(const M4Vec &k, int m) const {
394:   const double theta = k.Theta();
395:   const double phi   = k.Phi();
396:
397:   // const std::vector<double> e1 = {1.0, 0.0, 0.0};
398:   // const std::vector<double> e2 = {0.0, 1.0, 0.0};
399:
400:   // First define in \vec{z}-direction, then rotate spatial part such that
401:   // \vec{k} = R \vec{z}
402:
403:   // Normalization
404:   const double N = 1.0 / std::sqrt(2.0);
405:
406:   Tensor1<std::complex<double>, 4> e;
407:
408:   // plus minus to minus plus
409:   const double neg = -static_cast<double>(m);
410:   const double pos = static_cast<double>(m);
411:
412:   if (m == 1 || m == -1) {
413:     e(0) = 0.0;
414:     e(1) = (neg * std::cos(theta) * std::cos(phi) + zi * std::sin(phi)) * N;
415:     e(2) = (neg * std::cos(theta) * std::sin(phi) - zi * std::cos(phi)) * N;
```



```
416:     e(3) = pos * std::sin(theta) * N;
417:     return e;
418:   }
419:
420:   throw std::invalid_argument("MDirac::EpsSpin1: helicity m should be -1 or 1");
421: }
422:
423: // Polarization vector eps^{m}_\mu(k) for massive spin-1 with helicity m =
424: // -1,0,1
425: //
426: // m = +-1 state depends only on the direction of the momentum
427: // m = 0   state depends also on magnitude
428: //
429: // {t,x,y,z} order convention!
430: //
431: // Should obey sum:
432: // \sum_{\lambda = -1,0,1} \eps_\mu(k, \lambda) \eps_\nu(k, \lambda)*
433: // = -g_{\mu\nu} + k_\mu k_\nu / m^2
434: //
435: Tensor1<std::complex<double>, 4> MDirac::EpsMassiveSpin1(const M4Vec &k, int m) const {
436:   // \eps^{0},+1} (massless case applies here too)
437:   if (m == -1 || m == 1) { return EpsSpin1(k, m); }
438:   // Should be 0 at this point
439:   if (m != 0) {
440:     throw std::invalid_argument("MDirac::EpsMassiveSpin1: helicity m should be -1, 0 or 1");
441:   }
442:
443:   // \eps^{0},\mu} (longitudinal case)
444:   const double theta = k.Theta();
445:   const double phi   = k.Phi();
446:   const double E     = k.E();
447:   const double M     = k.M();
448:
449:   Tensor1<std::complex<double>, 4> e;
450:   e(0) = k.P3mod() / M;
451:   e(1) = E * std::sin(theta) * std::cos(phi) / M;
452:   e(2) = E * std::sin(theta) * std::sin(phi) / M;
453:   e(3) = E * std::cos(theta) / M;
454:
455:   return e;
456: }
457:
458: // Polarization tensor eps^{m}_{\mu\nu}(k) for massive spin-2 with helicity m =
459: // -2,-1,0,1,2
460: //
461: // {t,x,y,z} convention!
462: //
463: // Should obey sum: {eps_{\mu\nu}^{m}(k)}^* eps^{n}_{\mu\nu}(k) } = \delta_{mn}
464: //
465: Tensor2<std::complex<double>, 4, 4> MDirac::EpsMassiveSpin2(const M4Vec &k, int m) const {
466:   if (!(m == -2 || m == -1 || m == 0 || m == 1 || m == 2)) {
467:     throw std::invalid_argument("MDirac::EpsMassiveSpin2: m is not -2,-1,0,1,2");
468:   }
469:   Tensor2<std::complex<double>, 4, 4> epsmat;
470:
471:   const int offset = 1;  // array indexing (-1 means 0)
472:
473:   // Get Massive Spin-1 basis vectors
474:   std::array<Tensor1<std::complex<double>, 4>, 3> epsvec;
475:   for (const int &m : {-1, 0, 1}) { epsvec[m + offset] = EpsMassiveSpin1(k, m); }
476:
477:   // Loop over two Lorentz indices
478:   for (const auto &mu : LI) {
479:     for (const auto &nu : LI) {
480:       // Clebsch-Gordan decomposition
481:       for (const int &m1 : {-1, 0, 1}) {
482:         for (const int &m2 : {-1, 0, 1}) {
483:           // ClebschGordan(double j1, double j2, double m1, double m2,
484:           //  double j, double m)
485:           // Get coefficient <1,1, m1, m2| 2,m>
486:           epsmat(mu, nu) += gra::spin::ClebschGordan(1.0, 1.0, m1, m2, 2.0, m) *
487:                             epsvec[m1 + offset](mu) * epsvec[m2 + offset](nu);
488:         }
489:       }
490:     }
491:   }
492:   return epsmat;
493: }
494:
495: // Adjoint Dirac spinor: \bar{u} = u^dagger * gamma^0
496: std::vector<std::complex<double>> MDirac::Bar(
497:     const std::vector<std::complex<double>> &spinor) const {
498:   // First conjugate elements, then a matrix product with gamma^0 matrix
```



```
499:   return gra::matoper::VecMatMultiply(gra::matoper::VecDagger(spinor), gamma_up[0]);
500: }
501:
502: // Feynman slash matrix operator: \slash{a} = \gamma_\mu a^\mu = \gamma^\mu
503: // a_\mu
504: //
505: // Input assumed contravariant (upper) index 4-vector
506: MMatrix<std::complex<double>> MDirac::FSlash(const M4Vec &a) const {
507:   MMatrix<std::complex<double>> aslash(4, 4, 0.0);  // Init with zero!
508:   for (const auto &mu : LI) { aslash += gamma_up[mu] * (a % mu); }
509:   return aslash;
510: }
511:
512: // ------------------------------------------------------------------------
513: // High Energy (E >> m) relation for the vertex current when p = p' (e.g.
514: // forward scattering)
515: //
516: // \bar{u}_s'(p) \gamma^\mu u_s(p) = 2p^\mu \delta_{s',s}
517: // \bar{v}_s'(p) \gamma^\mu v_s(p) = 2p^\mu \delta_{s',s}
518: //
519: // Other relation:
520: // \bar{u}_s'(p) \gamma^0 v_s(-p) = 0
521: // \bar{v}_s'(p) \gamma^0 u_s(-p) = 0
522: //
523: double MDirac::TestSpinorHELimit(const M4Vec &pi, const M4Vec &pf) const {
524:   std::cout << "MDirac::TestSpinorHELimit:" << std::endl;
525:
526:   const M4Vec psum = pi + pf;
527:
528:   double ratiosum = 0.0;
529:
530:   // Helicities
531:   for (const auto &hi : {1, 2}) {
532:     const std::vector<std::complex<double>> u = uDirac(pi, hi);
533:
534:     // Helicities
535:     for (const auto &hf : {1, 2}) {
536:       const std::vector<std::complex<double>> ubar = Bar(uDirac(pf, hf));
537:
538:       if (hi != hf) {
539:         std::cout << std::endl;
540:         continue;
541:       }  // Helicity conservation
542:
543:       for (const auto &mu : LI) {
544:         for (const auto &nu : LI) {
545:           const std::vector<std::complex<double>> prod = (gamma_up[mu] * (psum % nu)) * u;
546:
547:           const std::complex<double> lhs = gra::matoper::VecVecMultiply(ubar, prod);
548:           const double               rhs = (psum % mu) * (psum % nu) * Delta(hi, hf);
549:
550:           const double absratio = std::abs(std::real(lhs)) / std::abs(rhs);
551:           ratiosum += absratio;
552:
553:           if (absratio < 0.9 || absratio > 1.1) {
554:             std::cout << rang::fg::red;
555:           } else {
556:             std::cout << rang::fg::green;
557:           }
558:
559:           printf(
560:               " hi = %2d, hf = %2d | {mu,nu} = {%1u,%1u} | Re[lhs] = "
561:               "%0.3E \t rhs = %0.3E \t |lhs| / |rhs| = %0.5f \n",
562:               hi, hf, mu, nu, std::real(lhs), rhs, absratio);
563:
564:           std::cout << rang::fg::reset;
565:         }
566:       }
567:     }
568:   }
569:
570:   std::cout << std::endl << std::endl;
571:   return ratiosum;
572: }
573:
574: // ------------------------------------------------------------------------
575: // [REFERENCE: https://arxiv.org/pdf/hep-ph/0110108.pdf]
576:
577: // Spinor product: s_\lambda(p1,p2)
578: std::complex<double> MDirac::sProd(const M4Vec &p1, const M4Vec &p2, int helicity) const {
579:   return gra::matoper::VecVecMultiply(Bar(uGauge(p1, helicity)), uGauge(p2, -helicity));
580: }
581:
```



```
582: // Helicity u-spinor via massless gauge vector
583: //
584: std::vector<std::complex<double>> MDirac::uGauge(const M4Vec &p, int helicity) const {
585:   const M4Vec l(100, 0, 0, 100);  // arbitrary "gauge vector"
586:   if (BASIS == "D") {                  // Dirac
587:     return ((FSlash(p) + I4 * p.M()) / msqrt(2.0 * (p * l))) * uHelDirac(1, -helicity);
588:   } else if (BASIS == "C") {  // Chiral
589:     return ((FSlash(p) + I4 * p.M()) / msqrt(2.0 * (p * l))) * uHelChiral(1, -helicity);
590:   }
591:   throw std::invalid_argument("MDirac::uGauge: Unknown BASIS = " + BASIS);
592: }
593:
594: // Helicity u-spinor via massless gauge vector
595: //
596: std::vector<std::complex<double>> MDirac::vGauge(const M4Vec &p, int helicity) const {
597:   const M4Vec l(100, 0, 0, 100);  // arbitrary "gauge vector"
598:   if (BASIS == "D") {                  // Dirac
599:     return (-(FSlash(p) - I4 * p.M()) / msqrt(2.0 * (p * l))) * vHelDirac(1, -helicity);
600:   } else if (BASIS == "C") {  // Chiral
601:     return (-(FSlash(p) - I4 * p.M()) / msqrt(2.0 * (p * l))) * vHelChiral(1, -helicity);
602:   }
603:   throw std::invalid_argument("MDirac::vGauge: Unknown BASIS = " + BASIS);
604: }
605:
606: // Construct spin-1/2 helicity spinors {-1,1} [indexing with 0,1]
607: std::array<std::vector<std::complex<double>>, 2> MDirac::SpinorStates(
608:     const M4Vec &p, const std::string &type) const {
609:   std::array<std::vector<std::complex<double>>, 2> spinor;
610:
611:   if (BASIS == "D") {  // Dirac
612:     for (const auto &m : {0, 1}) {
613:       if (type == "u") { spinor[m] = MDirac::uHelDirac(p, SPINORSTATE[m]); }
614:       if (type == "ubar") { spinor[m] = Bar(MDirac::uHelDirac(p, SPINORSTATE[m])); }
615:
616:       if (type == "v") { spinor[m] = MDirac::vHelDirac(p, SPINORSTATE[m]); }
617:       if (type == "vbar") { spinor[m] = Bar(MDirac::vHelDirac(p, SPINORSTATE[m])); }
618:     }
619:   } else if (BASIS == "C") {  // Chiral
620:     for (const auto &m : {0, 1}) {
621:       if (type == "u") { spinor[m] = MDirac::uHelChiral(p, SPINORSTATE[m]); }
622:       if (type == "ubar") { spinor[m] = Bar(MDirac::uHelChiral(p, SPINORSTATE[m])); }
623:
624:       if (type == "v") { spinor[m] = MDirac::vHelChiral(p, SPINORSTATE[m]); }
625:       if (type == "vbar") { spinor[m] = Bar(MDirac::vHelChiral(p, SPINORSTATE[m])); }
626:     }
627:   } else {
628:     throw std::invalid_argument("MDirac::SpinorStates: Unknown basis " + BASIS);
629:   }
630:
631:   return spinor;
632: }
633:
634: // Construct Massless Spin-1 polarization vectors (m = -1,1) [indexing with 0,1]
635: //
636: // Input p as contravariant (upper index)
637: //
638: std::array<Tensor1<std::complex<double>>, 4>, 2> MDirac::MasslessSpin1States(const M4Vec &      p,
639:                                                                               const std::string &type,
640:                                                                               bool INDEX_UP) const {
641:   std::array<Tensor1<std::complex<double>, 4>, 2> eps;
642:   const std::vector<int>                          lambda = {-1, 1};
643:
644:   for (const auto &m : indices(lambda)) {  // loop over helicities
645:     eps[m] = MDirac::EpsSpin1(p, lambda[m]);
646:
647:     if (type == "conj") {  // Take complex conjugate per element
648:       for (const auto &mu : LI) { eps[m](mu) = std::conj(eps[m](mu)); }
649:     }
650:     if (!INDEX_UP) {  // Return covariant (lower index) version
651:       for (const auto &mu : {1, 2, 3}) { eps[m](mu) = -eps[m](mu); }
652:     }
653:   }
654:   return eps;
655: }
656:
657: // Construct Massive Spin-1 polarization vectors (m = -1,0,1) [indexing with 0,1,2]
658: //
659: // Input p as contravariant (upper index)
660: //
661: std::array<Tensor1<std::complex<double>>, 4>, 3> MDirac::MassiveSpin1States(const M4Vec &      p,
662:                                                                              const std::string &type,
663:                                                                              bool INDEX_UP) const {
664:   std::array<Tensor1<std::complex<double>, 4>, 3> eps;
```

```
665:
666:   const std::vector<int> lambda = {-1, 0, 1};
667:
668:   for (const auto &m : indices(lambda)) {  // loop over helicities
669:     eps[m] = MDirac::EpsMassiveSpin1(p, lambda[m]);
670:
671:     if (type == "conj") {  // Take complex conjugate per element
672:       for (const auto &mu : LI) { eps[m](mu) = std::conj(eps[m](mu)); }
673:     }
674:     if (!INDEX_UP) {  // Return covariant (lower index) version
675:       for (const auto &mu : {1, 2, 3}) { eps[m](mu) = -eps[m](mu); }
676:     }
677:   }
678:   return eps;
679: }
680:
681: // ---------------------------------------------------------------------
682: // Test functions
683:
684: // Test gamma matrix anticommutation relation:
685: // {\gamma^\mu, \gamma^nu} = 2g^{mu\nu} I_4
686: //
687: double MDirac::TestGammaAntiCommutation() const {
688:   double diffsum = 0.0;
689:
690:   for (const auto &mu : LI) {
691:     for (const auto &nu : LI) {
692:       const MMatrix<std::complex<double>> AC_lo =
693:           gamma_lo[mu] * gamma_lo[nu] + gamma_lo[nu] * gamma_lo[mu];
694:       std::cout << "gamma_lo:: mu:" << mu << " nu: " << nu << std::endl;
695:       AC_lo.Print();
696:
697:       const MMatrix<std::complex<double>> AC_up =
698:           gamma_up[mu] * gamma_up[nu] + gamma_up[nu] * gamma_up[mu];
699:       std::cout << "gamma_up:: mu:" << mu << " nu: " << nu << std::endl;
700:       AC_up.Print();
701:
702:       diffsum += (AC_lo - AC_up).FrobNorm();
703:     }
704:   }
705:   return diffsum;
706: }
707:
708: // Test \slash{p}\slash{p} = p^2 I_4 (identity matrix being I4)
709: //
710: double MDirac::TestFSlashFSlash(const M4Vec &p) const {
711:   const MMatrix<std::complex<double>> A = FSlash(p) * FSlash(p);
712:   const MMatrix<std::complex<double>> B = I4 * p.M2();
713:
714:   const double norm = (A - B).FrobNorm();
715:   std::cout << "MDirac::TestFSlashFSlash:: Frobenius norm |A-B|_F: " << norm << std::endl;
716:   return norm;
717: }
718:
719: // Check Dirac u- and v-spinor normalization and completeness relations
720: //
721: // 1. Normalization of the Lorentz invariant inner product:
722: //
723: //    \bar{u}_s'(p) u_s(p) = +2m \delta_{s's}
724: //    \bar{v}_s'(p) v_s(p) = -2m \delta_{s's}
725: //    \bar{u}_s'(p) v_s(p) = 0
726: //    \bar{v}_s'(p) u_s(p) = 0
727: //
728: // Note that \dagger{u}_s'(p) u_s(p) = 2E \delta_{s's}
729: // (standard relativistic normalization of a wavefunction)
730: //
731: //
732: // 2. Completeness relation (Projection operators):
733: //
734: //    \sum_{s=+-} u_s(p) \bar{u}_s(p) = \slash{p} + m
735: //    \sum_{s=+-} v_s(p) \bar{v}_s(p) = \slash{p} - m
736: //
737: //
738: // Set type = "u" or "v"
739: //
740: // Normalization test (add to unit tests!)
741: //
742: // SPINOR      CHIRAL  |  DIRAC
743: // ------------------------------------------
744: // uDirac    | -       |  OK
745: // vDirac    | -       |  OK
746: // uHelDirac | -       |  OK
747: // vHelDirac | -       |  OK
```



```
748: // uHel      | OK      |  -
749: // vHel      | OK      |  -
750: //
751: // mode == "helicity" (default), "gauge", "spin"
752: //
753: double MDirac::TestSpinorComplete(const M4Vec &p, const std::string &type,
754:                                   const std::string &mode) const {
755:   std::cout << "MDirac::TestSpinorComplete: Type: " << type << std::endl;
756:   // InitGammaMatrices(basis);
757:   MMatrix<std::complex<double>> lhs(4, 4, 0.0);  // Init with zero!
758:   const double                  SIGN = ((type == "u") ? 1.0 : -1.0);
759:
760:   for (const auto &lambda : SPINORSTATE) {
761:     std::vector<std::complex<double>> spinor;
762:
763:     if (type == "u") {
764:       if (BASIS == "D") {
765:         spinor = uHelDirac(p, lambda);
766:
767:         if (mode == "spin") { spinor = uDirac(p, lambda); }
768:       }
769:       if (BASIS == "C") { spinor = uHelChiral(p, lambda); }
770:       if (mode == "gauge") { spinor = uGauge(p, lambda); }
771:
772:     } else if (type == "v") {
773:       if (BASIS == "D") {
774:         spinor = vHelDirac(p, lambda);
775:
776:         if (mode == "spin") { spinor = vDirac(p, lambda); }
777:       }
778:       if (BASIS == "C") { spinor = vHelChiral(p, lambda); }
779:
780:       if (mode == "gauge") { spinor = vGauge(p, lambda); }
781:
782:     } else {
783:       throw std::invalid_argument("MDirac::TestDiracSpinorComplete: Unknown type (set u or v)");
784:     }
785:     // Adjoint
786:     const std::vector<std::complex<double>> spinorbar = Bar(spinor);
787:
788:     // Check normalization
789:     const double nlhs =
790:         std::real(gra::matoper::VecVecMultiply(spinorbar, spinor));  // Take real to cast to double
791:     const double nrhs = SIGN * 2. * p.M();
792:
793:     printf("s = %2d : Normalization = ", lambda);
794:     if (gra::aux::AssertRatio(nlhs, nrhs, 5E-3)) {
795:       std::cout << rang::fg::green << nlhs << " | 2 x p.M = " << nrhs << " OK" << rang::fg::reset
796:                 << std::endl;
797:     } else {
798:       std::cout << rang::fg::red << nlhs << " | 2 x p.M = " << nrhs << " NOT OK" << rang::fg::reset
799:                 << std::endl;
800:     }
801:
802:     // Take outerproduct, sum
803:     lhs += gra::matoper::OuterProd(spinor, spinorbar);
804:   }
805:   std::cout << std::endl;
806:
807:   // Completeness relation
808:   const MMatrix<std::complex<double>> rhs = FSlash(p) + I4 * p.M() * SIGN;
809:
810:   // Compare
811:   lhs.Print("sum_{s}{spinor_s(p) barspinor_s(p)}");
812:   std::cout << std::endl;
813:   rhs.Print("slash(p) + I4*m*SIGN");
814:   std::cout << std::endl;
815:
816:   const double frobnorm = (lhs - rhs).FrobNorm();
817:   if (frobnorm < 0.1) {
818:     std::cout << rang::fg::green << "Completeness relation OK, Frobenius norm = " << frobnorm
819:               << rang::fg::reset << std::endl;
820:   } else {
821:     std::cout << rang::fg::red << "Completeness relation NOT OK, Frobenius norm = " << frobnorm
822:               << rang::fg::reset << std::endl;
823:   }
824:   std::cout << std::endl;
825:
826:   return frobnorm;
827: }
828:
829:
830: // Check massive spin1-polarization sum (completeness relation)
```



```
831: //
832: // \sum_{\lambda = -1,0.,1} eps^\mu(k,\lambda) eps^{\nu*)(k,\lambda)
833: // = -g^{\mu\nu} + k^\mu k^\nu / M^2
834: //
835: double MDirac::TestMassiveSpin1Complete(const M4Vec &k) const {
836:   double absdiffsum = 0.0;
837:
838:   for (const auto &mu : LI) {
839:     for (const auto &nu : LI) {
840:       // Massive spin-1 helicities
841:       std::complex<double> sum = 0.;
842:       for (const int &lambda : {-1, 0, 1}) {
843:         const Tensor1<std::complex<double>, 4> eps = EpsMassiveSpin1(k, lambda);
844:         sum += eps(mu) * std::conj(eps(nu));
845:       }
846:       const double lhs = std::real(sum);
847:
848:       // Right hand side
849:       const double rhs = -g[mu][nu] + k[mu] * k[nu] / k.M2();
850:       printf("MDirac::TestMassiveSpin1Complete: lhs = %0.2E, rhs = %0.2E \n", lhs, rhs);
851:
852:       absdiffsum += std::abs(lhs - rhs);
853:     }
854:   }
855:   return absdiffsum;
856: }
857:
858:
859: }  // namespace gra
860:
```



```
  1: // Factorized type phase space class
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: // C++
  7: #include <algorithm>
  8: #include <complex>
  9: #include <iostream>
 10: #include <random>
 11: #include <vector>
 12:
 13: // Own
 14: #include "Granitti/MAux.h"
 15: #include "Granitti/MFactorized.h"
 16: #include "Granitti/MForm.h"
 17: #include "Granitti/MFragment.h"
 18: #include "Granitti/MKinematics.h"
 19: #include "Granitti/MMath.h"
 20: #include "Granitti/MMatrix.h"
 21: #include "Granitti/MProcess.h"
 22: #include "Granitti/MSpin.h"
 23: #include "Granitti/MUserCuts.h"
 24:
 25: // Libraries
 26: #include "rang.hpp"
 27:
 28: using gra::aux::indices;
 29:
 30: using gra::math::CheckEMC;
 31: using gra::math::PI;
 32: using gra::math::abs2;
 33: using gra::math::msqrt;
 34: using gra::math::pow2;
 35: using gra::math::pow3;
 36: using gra::math::pow4;
 37: using gra::math::pow5;
 38: using gra::math::zi;
 39:
 40: using gra::PDG::GeV2barn;
 41:
 42: namespace gra {
 43:
 44: // This is needed by construction
 45: MFactorized::MFactorized() { Initialize(); }
 46:
 47: // Constructor
 48: MFactorized::MFactorized(std::string process, const std::vector<aux::OneCMD> &syntax) {
 49:   Initialize();
 50:   InitHistograms();
 51:   SetProcess(process, syntax);
 52:
 53:   // Init final states
 54:   M4Vec zerovec(0, 0, 0, 0);
 55:   for (std::size_t i = 0; i < 10; ++i) { lts.pfinal.push_back(zerovec); }
 56:   std::cout << "MFactorized:: [Constructor done]" << std::endl;
 57: }
 58:
 59: void MFactorized::Initialize() {
 60:   const std::vector<std::string> supported = {"PP", "yP", "OP", "yy", "gg"};
 61:   CID                                      = "F";
 62:   ProcPtr                                  = MSubProc(supported, CID);
 63: }
 64:
 65: // Destructor
 66: MFactorized::~MFactorized() {}
 67:
 68: // Initialize cut and process spesific postsetup
 69: void MFactorized::post_Constructor() {
 70:   // Set sampling boundaries
 71:   SetTechnicalBoundaries(gcuts, EXCITATION);
 72:
 73:   // Initialize phase space dimension
 74:   ProcPtr.LIPSDIM = 5 + 1;  // All processes, +1 from central system mass
 75:
 76:   if (EXCITATION == 1) { ProcPtr.LIPSDIM += 1; }
 77:   if (EXCITATION == 2) { ProcPtr.LIPSDIM += 2; }
 78: }
 79:
 80: // Update kinematics (screening kT loop calls this)
 81: // Exact 4-momentum conservation at loop vertices, that is, by using this one
 82: // does not assume vanishing external momenta in the screening loop calculation.
 83:
```

```
84: bool MFactorized::LoopKinematics(const std::vector<double> &p1p, const std::vector<double> &p2p) {
85:   static const M4Vec beamsum = lts.pbeam1 + lts.pbeam2;
86:   const unsigned int Nf      = lts.decaytree.size() + 2;  // Number of final states
87:
88:   // SET new final states pT degrees of freedom
89:   lts.pfinal[1].SetPxPy(p1p[0], p1p[1]);
90:   lts.pfinal[2].SetPxPy(p2p[0], p2p[1]);
91:
92:   const double pt1 = lts.pfinal[1].Pt();
93:   const double pt2 = lts.pfinal[2].Pt();
94:
95:   // Read original variables
96:   const double m1 = lts.pfinal_orig[1].M();
97:   const double m2 = lts.pfinal_orig[2].M();
98:
99:   const double mX = lts.pfinal_orig[0].M();
100:  const double yX = lts.pfinal_orig[0].Rap();
101:
102:  // Central system momentum
103:  lts.pfinal[0].SetPxPy(-(lts.pfinal[1].Px() + lts.pfinal[2].Px()),
104:                        -(lts.pfinal[1].Py() + lts.pfinal[2].Py()));
105:
106:  // Central system pz and E
107:  const double mtX = msqrt(pow2(mX) + lts.pfinal[0].Pt2());
108:  lts.pfinal[0].SetPzE(mtX * std::sinh(yX), mtX * std::cosh(yX));
109:
110:  // Energy overflow
111:  if (lts.pfinal[0].E() > (lts.sqrt_s - (m1 + m2))) { return false; }
112:
113:  double p1z =
114:      gra::kinematics::SolvePz(m1, m2, pt1, pt2, lts.pfinal[0].Pz(), lts.pfinal[0].E(), lts.s);
115:  double p2z = -(lts.pfinal[0].Pz() + p1z);  // by momentum conservation
116:  if (std::isnan(p1z)) { return false; }
117:
118:  // Enforce scattering direction +p -> +p, -p -> -p (VERY RARE POLYNOMIAL
119:  // BRANCH FLIP)
120:  if (p1z < 0 || p2z > 0) { return false; }
121:
122:  // Pz and E of protons/M*
123:  lts.pfinal[1].SetPzE(p1z, msqrt(pow2(m1) + pow2(pt1) + pow2(p1z)));
124:  lts.pfinal[2].SetPzE(p2z, msqrt(pow2(m2) + pow2(pt2) + pow2(p2z)));
125:
126:  // ------------------------------------------------------------------
127:  // Now boost if asymmetric beams
128:  if (std::abs(beamsum.Pz()) > 1e-6) {
129:    constexpr int sign = 1;  // positive -> boost to the lab
130:    kinematics::LorentzBoost(beamsum, lts.sqrt_s, lts.pfinal[1], sign);
131:    kinematics::LorentzBoost(beamsum, lts.sqrt_s, lts.pfinal[2], sign);
132:    kinematics::LorentzBoost(beamsum, lts.sqrt_s, lts.pfinal[0], sign);
133:  }
134:  // ------------------------------------------------------------------
135:
136:  if (!CheckEMC(beamsum - (lts.pfinal[1] + lts.pfinal[2] + lts.pfinal[0]))) { return false; }
137:
138:  return GetLorentzScalars(Nf);
139: }
140:
141: // Return Monte Carlo integrand weight
142: double MFactorized::EventWeight(const std::vector<double> &randvec, AuxIntData &aux) {
143:   double W = 0.0;
144:
145:   // Kinematics and cuts
146:   aux.kinematics_ok = B51RandomKin(randvec);
147:   aux.fiducts_ok    = FiducialCuts();
148:   aux.vetocuts_ok   = VetoCuts();
149:
150:   if (aux.Valid()) {
151:     // Matrix element squared
152:     const double MatESQ = GetAmp2();
153:
154:     // Calculate central system Phase Space volume
155:     double exact = 0.0;
156:     DecayWidthPS(exact);
157:     lts.DW_sum_exact.Add(gra::kinematics::MCW(exact, pow2(exact), 1), aux.vegasweight);
158:
159:     // Add to the integral sum and take into account the VEGAS weight
160:     lts.DW_sum.Add(lts.DW, aux.vegasweight);
161:
162:     double C_space = 1.0;
163:     if (lts.decaytree.size() != 0 && lts.PS_active) {  // We have some legs in the central system
164:       C_space = (lts.DW.Integral() / (2 * PI));        // /(2*PI) from phase space factorization
165:
166:       // ------------------------------------------------------------
```



```
167:       // Cascade resonances phase-space
168:       C_space *= CascadePS();
169:       // ------------------------------------------------------------
170:     }
171:
172:     // ** EVENT WEIGHT **
173:     W = C_space * (1.0 / S_factor) * B51PhaseSpaceWeight() * B51IntegralVolume() * MatESQ *
174:         GeV2barn / MollerFlux();
175:   }
176:
177:   aux.amplitude_ok = CheckInfNan(W);
178:
179:   // As the last STEP: Histograms
180:   const double totalweight = W * aux.vegasweight;
181:   FillHistograms(totalweight, lts);
182:
183:   return W;
184: }
185:
186: // Fiducial cuts
187: bool MFactorized::FiducialCuts() const { return CommonCuts(); }
188:
189: // Record event
190: bool MFactorized::EventRecord(HepMC3::GenEvent &evt) { return CommonRecord(evt); }
191:
192: void MFactorized::PrintInit(bool silent) const {
193:   if (!silent) {
194:     PrintSetup();
195:
196:     // Construct prettyprint diagram
197:     std::string proton1 = "----------EL--------->";
198:     std::string proton2 = "----------EL--------->";
199:
200:     if (EXCITATION == 1) { proton1 = "----------F2-xxxxxxxx>"; }
201:     if (EXCITATION == 2) {
202:       proton1 = "----------F2-xxxxxxxx>";
203:       proton2 = "----------F2-xxxxxxxx>";
204:     }
205:
206:     std::vector<std::string> feynmangraph;
207:     feynmangraph = {"||           ", "||           ", "xx--------->", "||           ", "||           "};
208:
209:     // Print diagram
210:     std::cout << proton1 << std::endl;
211:     for (const auto &i : indices(feynmangraph)) {
212:       if (SCREENING) {  // Put red
213:         std::cout << rang::fg::red << "     **     " << rang::style::reset;
214:       } else {
215:         std::cout << rang::fg::red << "            " << rang::style::reset;
216:       }
217:       std::cout << feynmangraph[i] << std::endl;
218:     }
219:     std::cout << proton2 << std::endl;
220:     std::cout << std::endl;
221:
222:     std::cout << std::endl;
223:     std::cout << rang::style::bold << "Generation cuts:" << rang::style::reset << std::endl
224:               << std::endl;
225:     printf(
226:         "~ Rap : System rapidity     [min, max] = [%0.2f, %0.2f]     \t(user) \n"
227:         "~ M   : System mass         [min, max] = [%0.2f, %0.2f] GeV \t(user) \n"
228:         "~ Pt  : Forward leg         [min, max] = [%0.2f, %0.2f] GeV "
229:         "\t(fixed/user) \n",
230:         gcuts.Y_min, gcuts.Y_max, gcuts.M_min, gcuts.M_max, gcuts.forward_pt_min,
231:         gcuts.forward_pt_max);
232:
233:     if (EXCITATION != 0) {
234:       printf(
235:           "~ Xi  : Forward leg (M^2/s) [min, max] = [%0.2E, %0.2E]     "
236:           "\t(fixed/user) \n",
237:           gcuts.XI_min, gcuts.XI_max);
238:     }
239:
240:     PrintFiducialCuts();
241:   }
242: }
243:
244: // 5+1-dimensional phase space vector initialization
245: bool MFactorized::B51RandomKin(const std::vector<double> &randvec) {
246:   // log-change of variables for pt
247:   const double u1 =
248:       std::log(gcuts.forward_pt_min + ZERO_EPS) +
249:       (std::log(gcuts.forward_pt_max) - std::log(gcuts.forward_pt_min + ZERO_EPS)) * randvec[0];
```



```
250:   const double u2 =
251:       std::log(gcuts.forward_pt_min + ZERO_EPS) +
252:       (std::log(gcuts.forward_pt_max) - std::log(gcuts.forward_pt_min + ZERO_EPS)) * randvec[1];
253:
254:   const double pt1 = std::exp(u1);
255:   const double pt2 = std::exp(u2);
256:
257:   const double phi1 = 2.0 * gra::math::PI * randvec[2];
258:   const double phi2 = 2.0 * gra::math::PI * randvec[3];
259:   const double yX   = gcuts.Y_min + (gcuts.Y_max - gcuts.Y_min) * randvec[4];
260:
261:   // Pick daughter masses, can fail due to off-shelliness, then
262:   // retry
263:   unsigned int       trials   = 0;
264:   const unsigned int MAXTRIAL = 1e5;
265:   while (true) {
266:     double M_sum = 0.0;
267:
268:     // ============================================================
269:     for (const auto &i : indices(lts.decaytree)) {
270:       GetOffShellMass(lts.decaytree[i], lts.decaytree[i].m_offshell);
271:       M_sum += lts.decaytree[i].m_offshell;
272:     }
273:     // ============================================================
274:
275:     // Apply absolute boundary conditions first
276:     const double MARGIN = 1e-4;  // GeV
277:     M_MIN               = M_sum + MARGIN;
278:     M_MAX               = lts.sqrt_s - (lts.pbeam1.M() + lts.pbeam2.M());
279:
280:     // Apply generator cuts
281:     M_MIN = std::max(M_MIN, gcuts.M_min);
282:     M_MAX = std::min(M_MAX, gcuts.M_max);
283:
284:     if (M_MIN < M_MAX) { break; }
285:     ++trials;
286:     if (trials > MAXTRIAL) {
287:       throw std::invalid_argument(
288:           "MFactorized::B51RandomKin: Infinite loop in kinematics. Check the "
289:           "decaymode and cuts!");
290:     }
291:   };
292:
293:   // Sample mass squared
294:   const double m2X = pow2(M_MIN) + (pow2(M_MAX) - pow2(M_MIN)) * randvec[5];
295:
296:   // Forward M* system masses
297:   std::vector<double> mvec;
298:   std::vector<double> rvec;
299:   if (EXCITATION == 1) { rvec = {randvec[6]}; }
300:   if (EXCITATION == 2) { rvec = {randvec[6], randvec[7]}; }
301:   SampleForwardMasses(mvec, rvec);
302:
303:   return B51BuildKin(pt1, pt2, phi1, phi2, yX, m2X, mvec[0], mvec[1]);
304: }
305:
306: // Build kinematics for 2->3 skeleton
307: bool MFactorized::B51BuildKin(double pt1, double pt2, double phi1, double phi2, double yX,
308:                               double m2X, double m1, double m2) {
309:   static const M4Vec beamsum = lts.pbeam1 + lts.pbeam2;
310:
311:   // Final state 4-momenta, set px,py first
312:   M4Vec p1(pt1 * std::cos(phi1), pt1 * std::sin(phi1), 0, 0);
313:   M4Vec p2(pt2 * std::cos(phi2), pt2 * std::sin(phi2), 0, 0);
314:   M4Vec pX(-(p1.Px() + p2.Px()), -(p1.Py() + p2.Py()), 0, 0);
315:
316:   // Central system pz and E
317:   const double mtX = msqrt(m2X + pX.Pt2());
318:   pX.SetPzE(mtX * std::sinh(yX), mtX * std::cosh(yX));
319:
320:   // Energy overflow
321:   if (pX.E() > (lts.sqrt_s - (m1 + m2))) { return false; }
322:
323:   double p1z = gra::kinematics::SolvePz(m1, m2, pt1, pt2, pX.Pz(), pX.E(), lts.s);
324:   double p2z = -(pX.Pz() + p1z);  // by momentum conservation
325:
326:   // Enforce scattering direction +p -> +p, -p -> -p (VERY RARE POLYNOMIAL
327:   // BRANCH FLIP)
328:   if (p1z < 0 || p2z > 0) { return false; }
329:
330:   // Pz and E of forward protons/M*
331:   p1.SetPzE(p1z, msqrt(pow2(m1) + pow2(pt1) + pow2(p1z)));
332:   p2.SetPzE(p2z, msqrt(pow2(m2) + pow2(pt2) + pow2(p2z)));
```



```
333:
334:   // ------------------------------------------------------------
335:   // Now boost if asymmetric beams
336:   if (std::abs(beamsum.Pz()) > 1e-6) {
337:     constexpr int sign = 1;  // positive -> boost to the lab
338:     kinematics::LorentzBoost(beamsum, lts.sqrt_s, p1, sign);
339:     kinematics::LorentzBoost(beamsum, lts.sqrt_s, p2, sign);
340:     kinematics::LorentzBoost(beamsum, lts.sqrt_s, pX, sign);
341:   }
342:   // ------------------------------------------------------------
343:
344:   // Save
345:   lts.pfinal[1] = p1;
346:   lts.pfinal[2] = p2;
347:   lts.pfinal[0] = pX;  // Central system
348:
349:   // ------------------------------------------------------------
350:   // Kinematic checks
351:
352:   // Total 4-momentum conservation
353:   if (!CheckEMC(beamsum - (lts.pfinal[1] + lts.pfinal[2] + lts.pfinal[0]))) { return false; }
354:
355:   // ====================================================================
356:   // Central system decay tree first branch kinematics set up here, the
357:   // rest is done recursively
358:   //
359:   // Collect decay product masses
360:   std::vector<double> masses;
361:   for (const auto &i : indices(lts.decaytree)) {
362:     // @@ Note, we need to take offshell masses here calculated first in B51RandomKin @@
363:     masses.push_back(lts.decaytree[i].m_offshell);
364:   }
365:   std::vector<M4Vec> products;
366:
367:   // false if amplitude has dependence on the final state legs (generic),
368:   // true if amplitude is a function of central system kinematics only (limited)
369:   const bool UNWEIGHT = !lts.PS_active;
370:
371:   gra::kinematics::MCW w;
372:   // 2-body
373:   if (lts.decaytree.size() == 2) {
374:     w = gra::kinematics::TwoBodyPhaseSpace(lts.pfinal[0], msqrt(m2X), masses, products, random);
375:     // 3-body
376:   } else if (lts.decaytree.size() == 3) {
377:     w = gra::kinematics::ThreeBodyPhaseSpace(lts.pfinal[0], msqrt(m2X), masses, products, UNWEIGHT,
378:                                              random);
379:     // N-body
380:   } else if (lts.decaytree.size() > 3) {
381:     w = gra::kinematics::NBodyPhaseSpace(lts.pfinal[0], msqrt(m2X), masses, products, UNWEIGHT,
382:                                          random);
383:   }
384:
385:   if (w.GetW() < 0) {
386:     return false;  // Kinematically impossible
387:   }
388:   lts.DW = w;
389:
390:   // Collect decay products
391:   const unsigned int offset = 3;
392:   for (const auto &i : indices(lts.decaytree)) {
393:     lts.decaytree[i].p4    = products[i];
394:     lts.pfinal[i + offset] = products[i];
395:   }
396:
397:   // Treat decaytree recursively
398:   for (const auto &i : indices(lts.decaytree)) {
399:     if (!ConstructDecayKinematics(lts.decaytree[i])) { return false; }
400:   }
401:
402:   // Forward system fragmentation
403:   if (vetocuts.active) {  // N.B. need to fragment already now!, if cuts active
404:     if (!CEPForwardFragment()) { return false; }
405:   }
406:
407:   // ====================================================================
408:   // Check that we are above mass threshold -> not necessary, this is
409:   // done in mass sampling function
410:   const unsigned int Nf = lts.decaytree.size() + 2;
411:   return GetLorentzScalars(Nf);
412: }
413:
414: // Calculate Pure Phase Space Decay Width (Volume)
415: void MFactorized::DecayWidthPS(double &exact) const {
```

```
416:   exact = 0;
417:   // Massive exact closed form for 2-body case
418:   if (lts.decaytree.size() == 2) {
419:     exact = gra::kinematics::PS2Massive(lts.m2, pow2(lts.decaytree[0].p4.M()),
420:                                         pow2(lts.decaytree[1].p4.M()));
421:   }
422:   // Massless case
423:   if (lts.decaytree.size() > 2) {
424:     exact = gra::kinematics::PSnMassless(lts.m2, lts.decaytree.size());
425:   }
426: }
427:
428: // 5-Dim Integral Volume \int_{M^2_MIN}^{M^2_MAX} { dM^2 } [phi1] x [phi2] x
429: // [pt1] x [pt2] x [y]
430: // Integral over central mass^2 is separate [phase-space factorization], but
431: // encapsulated here (+1 dimension)
432: double MFactorized::B51IntegralVolume() const {
433:   // Forward leg integration
434:   const double forward_volume = ForwardVolume();
435:
436:   return (pow2(M_MAX) - pow2(M_MIN)) * (gcuts.Y_max - gcuts.Y_min) * forward_volume;
437: }
438:
439: // 5-Dim phase space weight
440: double MFactorized::B51PhaseSpaceWeight() const {
441:   const double J =
442:       1.0 / std::abs(lts.pfinal[1].Px() / lts.pfinal[1].E() -
443:                      lts.pfinal[2].Px() / lts.pfinal[2].E());  // Jacobian, close to 0.5
444:
445:   const double factor = (1.0 / 2.0) * (1.0 / pow5(2.0 * gra::math::PI)) *
446:                         (lts.pfinal[1].Pt() / (2.0 * lts.pfinal[1].E())) *
447:                         (lts.pfinal[2].Pt() / (2.0 * lts.pfinal[2].E())) * J;
448:
449:   return factor;
450: }
451:
452: // For high mass limit kinematics, see e.g. [arxiv.org/pdf/hep-ph/9903279.pdf]
453:
454: }  // namespace gra
455:
```



```
  1: // "Metaclass" for Spherical Harmonics Expansion - contains
  2: // dataprocessing/Minuit/ROOT plotting functions.
  3: //
  4: // All mathematical spherical harmonic processing is separated to MSpherical namespace.
  5: //
  6: //
  7: // (c) 2017-2020 Mikael Mieskolainen
  8: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  9:
 10: // C++
 11: #include <complex>
 12: #include <iostream>
 13: #include <random>
 14: #include <regex>
 15: #include <string>
 16: #include <vector>
 17:
 18: // ROOT
 19: #include "TCanvas.h"
 20: #include "TColor.h"
 21: #include "TGraphErrors.h"
 22: #include "TH1.h"
 23: #include "TH2.h"
 24: #include "TLatex.h"
 25: #include "TLegend.h"
 26: #include "TLine.h"
 27: #include "TLorentzVector.h"
 28: #include "TMinuit.h"
 29: #include "TMultiGraph.h"
 30: #include "TProfile.h"
 31: #include "TStyle.h"
 32:
 33: // Own
 34: #include "Granitti/Analysis/MHarmonic.h"
 35: #include "Granitti/Analysis/MROOT.h"
 36: #include "Granitti/M4Vec.h"
 37: #include "Granitti/MAux.h"
 38: #include "Granitti/MKinematics.h"
 39: #include "Granitti/MMath.h"
 40: #include "Granitti/MPDG.h"
 41: #include "Granitti/MSpherical.h"
 42:
 43: // Eigen
 44: #include <Eigen/Dense>
 45:
 46: using gra::aux::indices;
 47: using gra::math::PI;
 48: using gra::math::msqrt;
 49: using gra::math::pow2;
 50: using gra::math::zi;
 51:
 52: namespace gra {
 53: // Constructor
 54: MHarmonic::MHarmonic() {}
 55:
 56: // Initialize
 57: void MHarmonic::Init(const HPARAM &hp) {
 58:   param = hp;
 59:   NCOEF = (param.LMAX + 1) * (param.LMAX + 1);
 60:
 61:   // ------------------------------------------------------------------
 62:   // SETUP parameters of the fit
 63:   ACTIVE.resize(NCOEF);  // Active moments
 64:   ACTIVENF = 0;
 65:
 66:   for (int l = 0; l <= param.LMAX; ++l) {
 67:     for (int m = -l; m <= l; ++m) {
 68:       const int index = gra::spherical::LinearInd(l, m);
 69:       ACTIVE[index]   = true;
 70:
 71:       // FIX ODD MOMENTS TO ZERO
 72:       if (param.REMOVEODD && ((l % 2) != 0)) { ACTIVE[index] = false; }
 73:       // FIX NEGATIVE M TO ZERO
 74:       if (param.REMOVENEGATIVEM && (m < 0)) { ACTIVE[index] = false; }
 75:       if (ACTIVE[index]) { ++ACTIVENF; }
 76:     }
 77:   }
 78:
 79:   // ------------------------------------------------------------------
 80:   std::cout << std::endl;
 81:   param.Print();
 82:   std::cout << std::endl;
 83:   // ------------------------------------------------------------------
```



```
 84:
 85:   // Init arrays used by Minuit function
 86:   t_lm      = std::vector<double>(NCOEF, 0.0);
 87:   t_lm_error = std::vector<double>(NCOEF, 0.0);
 88:   errmat    = MMatrix<double>(NCOEF, NCOEF, 0.0);
 89:   covmat    = MMatrix<double>(NCOEF, NCOEF, 0.0);
 90:
 91:   // Test functions
 92:   gra::spherical::TestSphericalIntegrals(param.LMAX);
 93:
 94:   std::cout << rang::fg::yellow
 95:             << "<Spherical Harmonic Based (costheta,phi)_r.f. "
 96:                "Decomposition and Efficiency inversion>"
 97:             << std::endl
 98:             << std::endl;
 99:   std::cout << "TERMINOLOGY:" << rang::fg::reset << std::endl;
100:   std::cout << "  (G) Generated == Events in the angular flat phase space (no "
101:                "cuts on final "
102:                "states, only on the system)"
103:             << std::endl;
104:   std::cout << "  (F) Fiducial  == Events within the strict fiducial "
105:                "(geometric-kinematic) "
106:                "final state phase space (cuts on final states)"
107:             << std::endl;
108:   std::cout << "  (D) Detector  == Events after the detector efficiency losses, selection AND "
109:                "fiducial cuts"
110:             << std::endl;
111:   std::cout << std::endl;
112:   std::cout << "(D) subset of (F) subset of (G) (this strict hierarchy might "
113:                "be violated in "
114:                "some special cases)"
115:             << std::endl;
116:   std::cout << std::endl;
117:   std::cout << "  The basic idea is to define (G) such that minimal extrapolation " << std::endl;
118:   std::cout << "  is required from the strict fiducial (geometric) phase space (F). " << std::endl;
119:   std::cout << "  Flatness requirement of (G) is strictly required to represent "
120:                "moments in "
121:                "an unmixed basis (non-flat phase space <=> geometric moment mixing)."
122:             << std::endl;
123:   std::cout << std::endl;
124:   std::cout << rang::fg::yellow << "EXAMPLE OF A FORMALLY VALID DEFINITION:" << rang::fg::reset
125:             << std::endl;
126:   std::cout << "  G = {|Y(system)| < 0.9}" << std::endl;
127:   std::cout << "  F = {|eta(pi)|   < 0.9 && pt(pi) > 0.1 GeV}" << std::endl;
128:   std::cout << std::endl << std::endl;
129:   std::cout << "Note also the rotation between inactive coefficients and "
130:                "active one, due to "
131:                "moment mixing."
132:             << std::endl;
133:   std::cout << std::endl << std::endl;
134:
135:   // pause(5);
136: }
137:
138: // Destructor
139: MHarmonic::~MHarmonic() {}
140:
141: bool MHarmonic::PrintLoop(const std::string &output) const {
142:   // Add here code to print output to files ...
143:
144:   return true;
145: }
146:
147: // Plot all figures
148: void MHarmonic::PlotAll(const std::string &outputpath) const {
149:   for (const auto &OBSERVABLE : {0, 1, 2}) {
150:     // **** EFFICIENCY DECOMPOSITION ****
151:     Plot1DEfficiency(OBSERVABLE, outputpath);
152:
153:     // **** ALGEBRAIC INVERSE MOMENTS ****
154:     PlotFigures(det, OBSERVABLE, "h_{Moments}[MPP]<det>", 33, outputpath);
155:     PlotFigures(fid, OBSERVABLE, "h_{Moments}[MPP]<fid>", 33, outputpath);
156:     PlotFigures(fla, OBSERVABLE, "h_{Moments}[MPP]<fla>", 33, outputpath);
157:
158:     // **** EXTENDED MAXIMUM LIKELIHOOD INVERSE MOMENTS ****
159:     if (param.EML) {
160:       PlotFigures(det, OBSERVABLE, "h_{Moments}[EML]<det>", 33, outputpath);
161:       PlotFigures(fid, OBSERVABLE, "h_{Moments}[EML]<fid>", 33, outputpath);
162:       PlotFigures(fla, OBSERVABLE, "h_{Moments}[EML]<fla>", 33, outputpath);
163:     }
164:   }
165:
166:   // 2D
```



```
167:   std::vector<std::vector<int>> OBSERVABLES = {{0, 1}, {0, 2}, {1, 2}});  // Different pairs
168:
169:   for (const auto &OBSERVABLE2 : OBSERVABLES) {
170:     // **** EFFICIENCY DECOMPOSITION ****
171:     PlotFigures2D(fla, OBSERVABLE2, "{Response}[FLAT_REFERENCE]<fla>", 17, outputpath);
172:     PlotFigures2D(fid, OBSERVABLE2, "{Response}[FIDUCIAL_ACCEPTANCE]<fid>", 33, outputpath);
173:     PlotFigures2D(det, OBSERVABLE2, "{Response}[ACCEPTANCE_x_EFFICIENCY]<det>", 29, outputpath);
174:
175:     // **** ALGEBRAIC INVERSE MOMENTS ****
176:     PlotFigures2D(fla, OBSERVABLE2, "{Moments}[MPP]<fla>", 17, outputpath);
177:     PlotFigures2D(fid, OBSERVABLE2, "{Moments}[MPP]<fid>", 33, outputpath);
178:     PlotFigures2D(det, OBSERVABLE2, "{Moments}[MPP]<det>", 29, outputpath);
179:
180:     // **** EXTENDED MAXIMUM LIKELIHOOD INVERSE MOMENTS ****
181:     if (param.EML) {
182:       PlotFigures2D(fla, OBSERVABLE2, "{Moments}[EML]<fla>", 17, outputpath);
183:       PlotFigures2D(fid, OBSERVABLE2, "{Moments}[EML]<fid>", 33, outputpath);
184:       PlotFigures2D(det, OBSERVABLE2, "{Moments}[EML]<det>", 29, outputpath);
185:     }
186:   }
187:
188:   const int OBSERVABLE = 0;
189:   PlotExpansion(fid, OBSERVABLE, "h_{Moments}[MPP]<fid>", 33, outputpath);
190:   Plot2DExpansion(fla, OBSERVABLE, "h_{Moments}[MPP]<fla>", 33, outputpath);
191:   if (param.EML) {
192:     PlotExpansion(fid, OBSERVABLE, "h_{Moments}[EML]<fid>", 33, outputpath);
193:     Plot2DExpansion(fla, OBSERVABLE, "h_{Moments}[EML]<fla>", 33, outputpath);
194:   }
195: }
196:
197: // Synthesized (costheta,phi) plots
198: //
199: void MHarmonic::Plot2DExpansion(
200:     const std::map<gra::spherical::Meta, MTensor<gra::spherical::SH>> &tensor,
201:     unsigned int OBSERVABLE, const std::string &TYPESTRING, int barcolor,
202:     const std::string &outputpath) const {
203:   // ------------------------------------------------------------------
204:   // Extract name strings
205:
206:   // find (string)
207:   std::smatch sma;
208:   std::regex_search(TYPESTRING, sma, std::regex(R"(\(.*?\))"));  // R"()" for Raw string literals
209:   std::string DATAMODE = sma[0];
210:   DATAMODE             = DATAMODE.substr(1, DATAMODE.size() - 2);
211:
212:   // find <string>
213:   std::smatch smb;
214:   std::regex_search(TYPESTRING, smb, std::regex(R"(\<.*?\>)"));  // R"()" for Raw string literals
215:   std::string SPACE = smb[0];
216:   SPACE             = SPACE.substr(1, SPACE.size() - 2);
217:
218:   // find [string]
219:   std::smatch smc;
220:   std::regex_search(TYPESTRING, smc, std::regex(R"(\[.*?\])"));  // R"()" for Raw string literals
221:   std::string ALGO = smc[0];
222:   ALGO             = ALGO.substr(1, ALGO.size() - 2);
223:
224:   // ------------------------------------------------------------------
225:   // f(cos(theta),phi; M) synthesis
226:
227:   std::size_t N = 50;
228:
229:   // Cos(theta) and phi
230:   std::vector<double> costheta;
231:   std::vector<double> phi;
232:
233:   std::size_t BINS = 0;
234:
235:   // TH2 for each
236:   std::vector<std::vector<TH2D *>> h2;
237:
238:   // Loop over fla
239:   std::size_t source_ind = 0;
240:   for (const auto &source : tensor) {
241:     // legendstrs[ind] = source.first.LEGEND;
242:     BINS = source.second.size(OBSERVABLE);
243:
244:     // Add histograms
245:     h2.push_back(std::vector<TH2D *>(BINS, NULL));
246:
247:     // Loop over observable
248:
249:     for (std::size_t bin = 0; bin < BINS; ++bin) {
```

```
250:       // Get x-axis point
251:       // const double value = grid[OBSERVABLE][bin].center();
252:
253:       // Set indices (0,0,0, ..., 0)
254:       std::vector<std::size_t> cell(grid.size(), 0);
255:       cell[OBSERVABLE] = bin;
256:
257:       // Synthesize distribution
258:       MMatrix<double> Z;
259:
260:       if (ALGO == "MPP") {
261:         Z = spherical::Y_real_synthesize(source.second(cell).t_lm_MPP, ACTIVE, N, costheta, phi,
262:                                          true);
263:       } else if (ALGO == "EML") {
264:         Z = spherical::Y_real_synthesize(source.second(cell).t_lm_EML, ACTIVE, N, costheta, phi,
265:                                          true);
266:       }
267:
268:       // Create histogram
269:       const double     EPS  = 1e-3;
270:       const std::string name = "h2_" + std::to_string(source_ind) + "_" + std::to_string(bin);
271:       h2[source_ind][bin]    = new TH2D(name.c_str(), "; cos #theta; #phi (rad)", N, -(1 + EPS),
272:                                         1 + EPS, N, -(math::PI + EPS), math::PI + EPS);
273:
274:       for (std::size_t i = 0; i < N; ++i) {
275:         for (std::size_t j = 0; j < N; ++j) {
276:           h2[source_ind][bin]->Fill(costheta[i], phi[j], Z[i][j]);
277:         }
278:       }
279:     }
280:
281:     ++source_ind;
282:   }  // loop over sources
283:
284:   // Loop over bins
285:   for (std::size_t bin = 0; bin < BINS; ++bin) {
286:     TCanvas *c1 = new TCanvas("c1", "c1", 200 + tensor.size() * 400, 500);  // horizontal, vertical
287:     c1->Divide(tensor.size(), 1, 0.002, 0.001);
288:
289:     std::size_t source_ind = 0;
290:     std::string FRAME;
291:     for (const auto &source : tensor) {
292:       c1->cd(source_ind + 1);
293:       c1->cd(source_ind + 1)->SetRightMargin(0.13);
294:       //
295:       FRAME = source.first.FRAME;
296:       h2[source_ind][bin]->SetTitle(source.first.LEGEND.c_str());
297:       //
298:       h2[source_ind][bin]->GetXaxis()->CenterTitle();
299:       h2[source_ind][bin]->GetZaxis()->SetRangeUser(0.1, 1.0);
300:       h2[source_ind][bin]->Draw("COLZ");
301:       //
302:
303:       // -----------------------------------------------------------
304:       // Draw Lorentz FRAME string on left
305:       TText *t2 = new TText(0.4, 0.85,
306:                             Form("%s : [%0.2f, %0.2f] %s", FRAME.c_str(), grid[OBSERVABLE][bin].min,
307:                                  grid[OBSERVABLE][bin].max, xlabels[OBSERVABLE].c_str()));
308:       t2->SetNDC();
309:       t2->SetTextAlign(22);
310:       t2->SetTextColor(kBlack);
311:       t2->SetTextFont(43);
312:       t2->SetTextSize(18);
313:       // t2->SetTextAngle(45);
314:       t2->Draw("same");
315:       // -----------------------------------------------------------
316:
317:       ++source_ind;
318:     }
319:     //
320:     aux::CreateDirectory("./figs");
321:     aux::CreateDirectory("./figs/harmonicfit");
322:     aux::CreateDirectory("./figs/harmonicfit/" + outputpath + "/synthesis");
323:     //
324:     const std::string subpath =
325:         SPACE + "_OBS_" + std::to_string(OBSERVABLE) + "_" + FRAME + "_" + ALGO;
326:     const std::string fullpath = "./figs/harmonicfit/" + outputpath + "/synthesis/" + subpath;
327:     aux::CreateDirectory(fullpath);
328:     c1->Print(Form("%s/%04i.pdf", fullpath.c_str(), bin));
329:     //
330:     delete c1;
331:   }
332:
```



```
333:   // Delete all histograms
334:   for (std::size_t i = 0; i < h2.size(); ++i) {
335:     for (std::size_t j = 0; j < h2[i].size(); ++j) { delete h2[i][j]; }
336:   }
337: }
338:
339: void GetLegendPosition2(unsigned int N, double &x1, double &x2, double &y1, double &y2,
340:                         const std::string &legendposition) {
341:   // North-East
342:   x1 = 0.63;
343:   x2 = x1 + 0.18;
344:   y1 = 0.75 - 0.01 * N;
345:
346:   // South-East
347:   if (legendposition.compare("southeast") == 0) { y1 = 0.10 - 0.01 * N; }
348:   y2 = y1 + 0.05 * N;  // Scale by the number of histograms
349: }
350:
351: // LM-Expansion plots as a function of observables
352: //
353: void MHarmonic::PlotFigures(
354:     const std::map<gra::spherical::Meta, MTensor<gra::spherical::SH>> &tensor,
355:     unsigned int OBSERVABLE, const std::string &TYPESTRING, int barcolor,
356:     const std::string &outputpath) const {
357:   // -----------------------------------------------------------------------
358:   std::shared_ptr<TCanvas> c1;
359:   gra::rootstyle::AutoGridCanvas(c1, ACTIVENDF);
360:   // -----------------------------------------------------------------------
361:
362:   if (OBSERVABLE > xlabels.size() - 1) {
363:     throw std::invalid_argument("MHarmonic::PlotFigures: Unknown observable " +
364:                                 std::to_string(OBSERVABLE));
365:   }
366:   const std::string xlabel = xlabels[OBSERVABLE];
367:
368:   // -----------------------------------------------------------------------
369:   // Extract name strings
370:
371:   // find [string]
372:   std::smatch sma;
373:   std::regex_search(TYPESTRING, sma, std::regex(R"(\[.*?\])"));  // R"()" for Raw string literals
374:   std::string DATAMODE = sma[0];
375:   DATAMODE             = DATAMODE.substr(1, DATAMODE.size() - 2);
376:
377:   // find <string>
378:   std::smatch smb;
379:   std::regex_search(TYPESTRING, smb, std::regex(R"(\<.*?\>)"));  // R"()" for Raw string literals
380:   std::string SPACE = smb[0];
381:   SPACE             = SPACE.substr(1, SPACE.size() - 2);
382:
383:   // find [string]
384:   std::smatch smc;
385:   std::regex_search(TYPESTRING, smc, std::regex(R"(\[.*?\])"));  // R"()" for Raw string literals
386:   std::string ALGO = smc[0];
387:   ALGO             = ALGO.substr(1, ALGO.size() - 2);
388:
389:   // -----------------------------------------------------------------------
390:
391:   // Loop over data sources
392:   std::size_t BINS = 0;
393:   int         ind  = 0;
394:
395:   // Graphs for each data [source] x [lm-moment]
396:   std::vector<std::vector<TGraphErrors *>> gr(tensor.size(),
397:                                               std::vector<TGraphErrors *>(ACTIVENDF, NULL));
398:
399:   // Legend titles
400:   std::vector<std::string> legendstrs(tensor.size());
401:
402:   // Minimum and maximum y-values for each plot
403:   std::vector<double> MINVAL(ACTIVENDF, 1e32);
404:   std::vector<double> MAXVAL(ACTIVENDF, -1e32);
405:
406:   // Turn of horizontal errors
407:   gStyle->SetErrorX(0);
408:
409:   std::vector<TMultiGraph *> mg(ACTIVENDF, NULL);
410:   for (std::size_t k = 0; k < ACTIVENDF; ++k) { mg[k] = new TMultiGraph(); }
411:
412:   std::string              FRAME;
413:   std::vector<std::string> TITLES;
414:
415:   // Y-axis title
```



```
416:   std::string yaxis_label = "Events / bin";
417:
418:   for (const auto &source : tensor) {
419:     legendstrs[ind] = source.first.LEGEND;
420:     BINS            = source.second.size(OBSERVABLE);
421:
422:     // Loop over moments
423:     int k = 0;
424:
425:     double SCALE = source.first.SCALE;
426:
427:     if (SCALE < 0 && source.first.YAXIS == "") { yaxis_label = "Normalized to 1"; }
428:     if (source.first.YAXIS != "") { yaxis_label = source.first.YAXIS; }
429:
430:     for (int l = 0; l <= param.LMAX; ++l) {
431:       for (int m = -l; m <= l; ++m) {
432:         const int index = gra::spherical::LinearInd(l, m);
433:
434:         if (!ACTIVE[index]) { continue; }  // Not active
435:
436:         // Set canvas position
437:         c1->cd(k + 1);
438:
439:         // Loop over observable
440:         double x[BINS]     = {0.0};
441:         double y[BINS]     = {0.0};
442:         double x_err[BINS] = {0.0};
443:         double y_err[BINS] = {0.0};
444:
445:         for (std::size_t bin = 0; bin < BINS; ++bin) {
446:           // Get x-axis point
447:           x[bin] = grid[OBSERVABLE][bin].center();
448:
449:           // Set indices (0,0,0, ..., 0)
450:           std::vector<std::size_t> cell(grid.size(), 0);
451:           cell[OBSERVABLE] = bin;
452:
453:           // CHOOSE DATAMODE
454:           if (ALGO == "MPP") {
455:             y[bin]     = source.second(cell).t_lm_MPP[index];
456:             y_err[bin] = source.second(cell).t_lm_MPP_error[index];
457:           } else if (ALGO == "EML") {
458:             y[bin]     = source.second(cell).t_lm_EML[index];
459:             y_err[bin] = source.second(cell).t_lm_EML_error[index];
460:           } else {
461:             throw std::invalid_argument(
462:                 "MHarmonic::PlotFigures: Unknown input: "
463:                 "DATAMODE = " +
464:                 DATAMODE + " ALGO = " + ALGO);
465:           }
466:         }
467:
468:         // -----------------------------------------------------
469:         // Scaling (e.g. luminosity)
470:         // Normalize lm = 00 to sum to 1 -> then apply to all
471:         if ((l == 0 && m == 0) && SCALE < 0.0) {
472:           double sum = 0.0;
473:           for (std::size_t bin = 0; bin < BINS; ++bin) { sum += y[bin]; }
474:           if (sum > 0) { SCALE = 1.0 / sum; }
475:         }
476:
477:         // Apply scale by user
478:         for (std::size_t bin = 0; bin < BINS; ++bin) {
479:           y[bin] *= SCALE;
480:           y_err[bin] *= SCALE;
481:         }
482:         // -----------------------------------------------------
483:
484:         // Save maximum for visualization
485:         for (std::size_t bin = 0; bin < BINS; ++bin) {
486:           MAXVAL[k] = y[bin] > MAXVAL[k] ? y[bin] : MAXVAL[k];
487:           MINVAL[k] = y[bin] < MINVAL[k] ? y[bin] : MINVAL[k];
488:         }
489:         // Data displayed using TGraphErrors
490:         gr[ind][k] = new TGraphErrors(BINS, x, y, x_err, y_err);
491:
492:         // Colors
493:         gr[ind][k]->SetMarkerColor(colors[ind]);
494:         gr[ind][k]->SetLineColor(colors[ind]);
495:         gr[ind][k]->SetFillColor(colors[ind]);
496:         gr[ind][k]->SetLineWidth(2.0);
497:         gr[ind][k]->SetMarkerStyle(21);  // square
498:         gr[ind][k]->SetMarkerSize(0.3);
```



```
499:
500:         FRAME  = source.first.FRAME;
501:         TITLES = source.first.TITLES;
502:
503:         // Add to the multigraph
504:         mg[k]->Add(gr[ind][k]);
505:
506:         ++k;
507:
508:       }  // over m
509:     }    // over l
510:
511:     ++ind;
512:   }  // Loop over sources
513:
514:   // Draw multigraph
515:   unsigned int k = 0;
516:
517:   // Aux variables
518:   std::shared_ptr<TPad>   tpad;
519:   std::shared_ptr<TLatex> l1;
520:   std::shared_ptr<TLatex> l2;
521:
522:   for (int l = 0; l <= param.LMAX; ++l) {
523:     for (int m = -l; m <= l; ++m) {
524:       const int index = gra::spherical::LinearInd(l, m);
525:
526:       if (!ACTIVE[index]) { continue; }  // Not active
527:
528:       // Set canvas position
529:       c1->cd(k + 1);
530:
531:       // First draw, then setup (otherwise crash)
532:       mg[k]->Draw("AC*");
533:
534:       // Title and y-axis
535:       if (k == 0) {
536:         // Title
537:         if (SPACE == "det") {
538:           mg[k]->SetTitle(Form("%s | #it{lm} = <%d,%d>", TITLES[0].c_str(), l, m));
539:         }
540:         if (SPACE == "fid") {
541:           mg[k]->SetTitle(Form("%s | #it{lm} = <%d,%d>", TITLES[1].c_str(), l, m));
542:         }
543:         if (SPACE == "fia") {
544:           mg[k]->SetTitle(Form("%s | #it{lm} = <%d,%d>", TITLES[2].c_str(), l, m));
545:         }
546:
547:         // y-axis (for some ROOT reason, this needs to be always after SetTitle)
548:         mg[k]->GetYaxis()->SetTitle(yaxis_label.c_str());
549:         gPad->SetLeftMargin(0.15);  // 15 per cent of pad for left margin, default is 10%
550:         mg[k]->GetYaxis()->SetTitleOffset(1.25);
551:
552:       } else {
553:         mg[k]->SetTitle(Form("Moment #it{lm} = <%d,%d>", l, m));
554:       }
555:
556:       // Set x-axis
557:       mg[k]->GetXaxis()->SetTitle(xlabel.c_str());
558:       // gStyle->SetBarWidth(0.5);
559:       // mg[k]->SetFillStyle(0);
560:
561:       mg[k]->GetXaxis()->SetTitleSize(0.05);
562:       mg[k]->GetXaxis()->SetLabelSize(0.05);
563:
564:       mg[k]->GetYaxis()->SetTitleSize(0.05);
565:       mg[k]->GetYaxis()->SetLabelSize(0.05);
566:
567:       // Set y-axis
568:       // mg[k]->GetYaxis()->SetTitle("Intensity");
569:
570:       gStyle->SetTitleFontSize(0.08);
571:
572:       if (k == 0) {
573:         gStyle->SetTitleW(0.95);  // width percentage
574:       }
575:
576:       // Y-axis range
577:       if (k == 0) {
578:         mg[k]->GetHistogram()->SetMaximum(MAXVAL[k] * 1.1);
579:         mg[k]->GetHistogram()->SetMinimum(0.0);
580:       } else {
581:         // Skip the first element (lm=00)
```

```
582:       const double maxval = *std::max_element(std::begin(MAXVAL) + 1, std::end(MAXVAL));
583:       const double minval = *std::min_element(std::begin(MINVAL) + 1, std::end(MINVAL));
584:       const double bound  = std::max(std::abs(minval), std::abs(maxval));
585:
586:       mg[k]->GetHistogram()->SetMaximum(bound * 1.1);
587:       mg[k]->GetHistogram()->SetMinimum(-bound * 1.1);
588:     }
589:
590:     // ------------------------------------------------------------
591:     // Draw Lorentz FRAME string on left
592:     TText *t2 = new TText(0.225, 0.825, FRAME.c_str());
593:     t2->SetNDC();
594:     t2->SetTextAlign(22);
595:     t2->SetTextColor(kRed + 2);
596:     t2->SetTextFont(43);
597:     t2->SetTextSize(std::ceil(1.0 / msqrt(ACTIVENDF)) * 16);
598:     // t2->SetTextAngle(45);
599:     t2->Draw("same");
600:     // ------------------------------------------------------------
601:
602:     // ------------------------------------------------------------
603:     // Draw horizontal line
604:     if (k != 0) {
605:       TLine *line = new TLine(grid[OBSERVABLE][0].min, 0.0,
606:                               grid[OBSERVABLE][grid[OBSERVABLE].size() - 1].max, 0.0);
607:       line->SetLineColor(kBlack);
608:       line->SetLineWidth(1.0);
609:       line->Draw("same");
610:     }
611:     // ------------------------------------------------------------
612:
613:     // ------------------------------------------------------------
614:     // Who made it
615:     if (k == ACTIVENDF - 1) {
616:       // New pad on top of all
617:       c1->cd();  // Important!
618:       gra::rootstyle::TransparentPad(tpad);
619:
620:       const double xpos = 0.99;
621:       gra::rootstyle::MadeInFinland(l1, l2, xpos);
622:     }
623:     // ------------------------------------------------------------
624:
625:     ++k;
626:   }
627: }
628:
629: // ------------------------------------------------------------
630: // Draw legend to the upper left most
631: c1->cd(1);
632:
633: // Create legend
634: double                x1, x2, y1, y2 = 0.0;
635: const std::string legendposition = "northeast";
636: GetLegendPosition2(tensor.size(), x1, x2, y1, y2, legendposition);
637: TLegend *legend = new TLegend(x1, y1, x2, y2);
638: legend->SetFillColor(0);   // White background
639: legend->SetBorderSize(0);  // No box
640: legend->SetTextSize(0.035);
641:
642: // Add legend entries
643: for (const auto &i : indices(gr)) { legend->AddEntry(gr[i][0], legendstrs[i].c_str()); }
644:
645: // Draw legend
646: legend->Draw("same");
647: // ------------------------------------------------------------
648:
649: // Require that we have data
650: if (BINS > 1) {
651:   aux::CreateDirectory("./figs");
652:   aux::CreateDirectory("./figs/harmonicfit");
653:   aux::CreateDirectory("./figs/harmonicfit/" + outputpath);
654:   const std::string subpath  = "OBS_" + std::to_string(OBSERVABLE) + "_" + FRAME;
655:   const std::string fullpath = "./figs/harmonicfit/" + outputpath + "/" + subpath;
656:   aux::CreateDirectory(fullpath);
657:   c1->Print(Form("%s/%s.pdf", fullpath.c_str(), TYPESTRING.c_str()));
658:
659:   // Merge pdfs using Ghostscript (gs)
660:   const std::string cmd = "gs -dBATCH -dNOPAUSE -q -sDEVICE=pdfwrite -sOutputFile=" + fullpath +
661:                           "/" + FRAME + "_merged.pdf " + fullpath + "/h*.pdf";
662:   if (system(cmd.c_str()) == -1) {
663:     throw std::invalid_argument("Error: Problem executing Ghostscript merge on pdfs!");
664:   }
```



```
665:   }
666:
667:   /*
668:   for (std::size_t i = 0; i < gr.size(); ++i) {
669:                 delete gr[i];
670:   }
671:   */
672: }
673:
674: // LM-Efficiency figures
675: //
676: void MHarmonic::PlotIDEfficiency(unsigned int OBSERVABLE, const std::string &outputpath) const {
677:   // ------------------------------------------------------------
678:   std::shared_ptr<TCanvas> c1;
679:   gra::rootstyle::AutoGridCanvas(c1, ACTIVENDF);
680:   // ------------------------------------------------------------
681:
682:   if (OBSERVABLE > xlabels.size() - 1) {
683:     throw std::invalid_argument("MHarmonic::PlotFigures: Unknown observable " +
684:                                 std::to_string(OBSERVABLE));
685:   }
686:   const std::string xlabel = xlabels[OBSERVABLE];
687:
688:   // Graphs for each efficiency [level] x [lm-moment]
689:   std::vector<std::vector<TGraphErrors *>> gr(3, std::vector<TGraphErrors *>(ACTIVENDF, NULL));
690:
691:   // Legend titles
692:   std::vector<std::string> legendstrs(3);
693:
694:   // Minimum and maximum y-values for each plot
695:   std::vector<double> MINVAL(ACTIVENDF, 1e32);
696:   std::vector<double> MAXVAL(ACTIVENDF, -1e32);
697:
698:   // Turn of horizontal errors
699:   gStyle->SetErrorX(0);
700:
701:   std::vector<TMultiGraph *> mg(ACTIVENDF, NULL);
702:   for (std::size_t k = 0; k < ACTIVENDF; ++k) { mg[k] = new TMultiGraph(); }
703:
704:   std::size_t BINS = det_DET.size(OBSERVABLE);
705:
706:   // Read data
707:   std::string                FRAME;
708:   std::vector<std::string> TITLES;
709:
710:   for (const auto &source : det) {
711:     FRAME  = source.first.FRAME;
712:     TITLES = source.first.TITLES;
713:     break;
714:   }
715:
716:   for (std::size_t level = 0; level < 3; ++level) {
717:     // Loop over moments
718:     int k = 0;
719:     for (int l = 0; l <= param.LMAX; ++l) {
720:       for (int m = -l; m <= l; ++m) {
721:         const int index = gra::spherical::LinearInd(l, m);
722:
723:         if (!ACTIVE[index]) { continue; }  // Not active
724:
725:         // Set canvas position
726:         c1->cd(k + 1);
727:
728:         // Loop over mass
729:         double x[BINS]     = {0.0};
730:         double y[BINS]     = {0.0};
731:         double x_err[BINS] = {0.0};
732:         double y_err[BINS] = {0.0};
733:
734:         for (std::size_t bin = 0; bin < BINS; ++bin) {
735:           // Get x-axis point
736:           x[bin] = grid[OBSERVABLE][bin].center();
737:
738:           // Set indices (0,0,0, ..., 0)
739:           std::vector<std::size_t> cell(grid.size(), 0);
740:           cell[OBSERVABLE] = bin;
741:
742:           // CHOOSE DATAMODE
743:           if (level == 0) {
744:             y[bin]     = det_DET(cell).E_lm[index];
745:             y_err[bin] = det_DET(cell).E_lm_error[index];
746:           } else if (level == 1) {
747:             y[bin]     = fid_DET(cell).E_lm[index];
```



```
748:             y_err[bin] = fid_DET(cell).E_lm_error[index];
749:           } else if (level == 2) {
750:             y[bin]     = fla_DET(cell).E_lm[index];
751:             y_err[bin] = fla_DET(cell).E_lm_error[index];
752:           }
753:
754:           // Save maximum for visualization
755:           MAXVAL[k] = y[bin] > MAXVAL[k] ? y[bin] : MAXVAL[k];
756:           MINVAL[k] = y[bin] < MINVAL[k] ? y[bin] : MINVAL[k];
757:         }
758:
759:         // Data displayed using TGraphErrors
760:         gr[level][k] = new TGraphErrors(BINS, x, y, x_err, y_err);
761:
762:         // Colors
763:         gr[level][k]->SetMarkerColor(colors[level]);
764:         gr[level][k]->SetLineColor(colors[level]);
765:         gr[level][k]->SetFillColor(colors[level]);
766:         gr[level][k]->SetLineWidth(2.0);
767:         gr[level][k]->SetMarkerStyle(21);  // square
768:         gr[level][k]->SetMarkerSize(0.3);
769:
770:         // Add to the multigraph
771:         mg[k]->Add(gr[level][k]);
772:
773:         ++k;
774:
775:       }  // over m
776:     }    // over l
777:
778:   }  // Loop over level
779:
780:   // Draw multigraph
781:   unsigned int k = 0;
782:
783:   std::shared_ptr<TPad>   tpad;
784:   std::shared_ptr<TLatex> l1;
785:   std::shared_ptr<TLatex> l2;
786:
787:   for (int l = 0; l <= param.LMAX; ++l) {
788:     for (int m = -l; m <= l; ++m) {
789:       const int index = gra::spherical::LinearInd(l, m);
790:
791:       if (!ACTIVE[index]) { continue; }  // Not active
792:
793:       // Set canvas position
794:       c1->cd(k + 1);
795:
796:       // First draw, then setup (otherwise crash)
797:       mg[k]->Draw("AC*");
798:
799:       // Title
800:       if (k == 0) {
801:         mg[k]->SetTitle(Form("Acceptance decomposition: #it{lm} = <%d,%d>", l, m));
802:       } else {
803:         mg[k]->SetTitle(Form("Moment #it{lm} = <%d,%d>", l, m));
804:       }
805:
806:       // Set x-axis
807:       mg[k]->GetXaxis()->SetTitle(xlabel.c_str());
808:       // gStyle->SetBarWidth(0.5);
809:       // mg[k]->SetFillStyle(0);
810:
811:       mg[k]->GetXaxis()->SetTitleSize(0.05);
812:       mg[k]->GetXaxis()->SetLabelSize(0.05);
813:
814:       mg[k]->GetYaxis()->SetTitleSize(0.05);
815:       mg[k]->GetYaxis()->SetLabelSize(0.05);
816:
817:       // Set y-axis
818:       // mg[k]->GetYaxis()->SetTitle(""");
819:
820:       gStyle->SetTitleFontSize(0.08);
821:
822:       if (k == 0) {
823:         gStyle->SetTitleW(0.95);  // width percentage
824:       }
825:       // Y-axis range
826:       if (k == 0) {
827:         mg[k]->GetHistogram()->SetMaximum(1.2);
828:         mg[k]->GetHistogram()->SetMinimum(0.0);
829:       } else {
830:         // Skip the first element (lm=00)
```



```
831:       const double maxval = *std::max_element(std::begin(MAXVAL) + 1, std::end(MAXVAL));
832:       const double minval = *std::min_element(std::begin(MINVAL) + 1, std::end(MINVAL));
833:       const double bound  = std::max(std::abs(minval), std::abs(maxval));
834:
835:       mg[k]->GetHistogram()->SetMaximum(bound * 1.1);
836:       mg[k]->GetHistogram()->SetMinimum(-bound * 1.1);
837:     }
838:
839:     // Draw Lorentz FRAME string on left
840:     TText *t2 = new TText(0.225, 0.825, FRAME.c_str());
841:     t2->SetNDC();
842:     t2->SetTextAlign(22);
843:     t2->SetTextColor(kRed + 2);
844:     t2->SetTextFont(43);
845:     t2->SetTextSize(std::ceil(1.0 / msqrt(ACTIVENDF)) * 16);
846:     // t2->SetTextAngle(45);
847:     t2->Draw("same");
848:
849:     // Draw horizontal line
850:     if (k != 0) {
851:       TLine *line = new TLine(grid[OBSERVABLE][0].min, 0.0,
852:                               grid[OBSERVABLE][grid[OBSERVABLE].size() - 1].max, 0.0);
853:       line->SetLineColor(kBlack);
854:       line->SetLineWidth(1.0);
855:       line->Draw("same");
856:     }
857:
858:     // Who made it
859:     if (k == ACTIVENDF - 1) {
860:       // New pad on top of all
861:       c1->cd();  // Important!
862:       gra::rootstyle::TransparentPad(tpad);
863:
864:       const double xpos = 0.99;
865:       gra::rootstyle::MadeInFinland(l1, l2, xpos);
866:     }
867:     ++k;
868:   }
869: }
870:
871: // ------------------------------------------------------------
872: // Draw legend to the upper left most
873:
874: c1->cd(1);
875:
876: // Create legend
877: double                x1, x2, y1, y2 = 0.0;
878: const std::string legendposition = "southeast";
879: GetLegendPosition2(3, x1, x2, y1, y2, legendposition);
880: TLegend *legend = new TLegend(x1 - 0.3, y1 + 0.1, x2 - 0.3, y2 + 0.1);
881: legend->SetFillColor(0);   // White background
882: legend->SetBorderSize(0);  // No box
883: legend->SetTextSize(0.04);
884:
885: // Add legend entries
886: legend->AddEntry(gr[0][0], TITLES[0].c_str());
887: legend->AddEntry(gr[1][0], TITLES[1].c_str());
888: legend->AddEntry(gr[2][0], TITLES[2].c_str());
889:
890: // Draw legend
891: legend->Draw("same");
892:
893: // ------------------------------------------------------------
894:
895: // Require that we have data
896: if (BINS > 1) {
897:   aux::CreateDirectory("./figs");
898:   aux::CreateDirectory("./figs/harmonicfit");
899:   aux::CreateDirectory("./figs/harmonicfit/" + outputpath);
900:   const std::string subpath  = "OBS_" + std::to_string(OBSERVABLE) + "_" + FRAME;
901:   const std::string fullpath = "./figs/harmonicfit/" + outputpath + "/" + subpath;
902:   aux::CreateDirectory(fullpath);
903:   c1->Print(Form("%s/h_Response.pdf", fullpath.c_str()));
904: }
905:
906: /*
907: for (std::size_t i = 0; i < gr.size(); ++i) {
908:               delete gr[i];
909: }
910: */
911: }
912:
913: // Print 2D-figures
```

```
914: void MHarmonic::PlotFigures2D(
915:     const std::map<gra::spherical::Meta, MTensor<gra::spherical::SH> > &tensor,
916:     const std::vector<int> &OBSERVABLE2, const std::string &TYPESTRING, int barcolor,
917:     const std::string &outputpath) const {
918:   /* Implement here ... */
919: }
920:
921: // Loop over system mass, pt, rapidity
922: //
923: void MHarmonic::HyperLoop(void (*fitfunc)(int &, double *, double &, double *, int),
924:                           const std::vector<gra::spherical::Omega> &MC,
925:                           const std::vector<gra::spherical::Data> &DATA, const HPARAM &hp) {
926:   // Initialize detector expansion tensors
927:   fla_DET = MTensor<gra::spherical::SH_DET>(
928:       {(unsigned int)hp.M[0], (unsigned int)hp.PT[0], (unsigned int)hp.Y[0]});
929:   fid_DET = fla_DET;
930:   det_DET = fla_DET;
931:
932:   // ------------------------------------------------------------
933:   // Create grid discretization
934:
935:   grid.resize(3);
936:   grid[0].resize(hp.M[0]);
937:   grid[1].resize(hp.PT[0]);
938:   grid[2].resize(hp.Y[0]);
939:
940:   // Mass steps
941:   const double M_STEP  = (hp.M[2] - hp.M[1]) / hp.M[0];
942:   const double PT_STEP = (hp.PT[2] - hp.PT[1]) / hp.PT[0];
943:   const double Y_STEP  = (hp.Y[2] - hp.Y[1]) / hp.Y[0];
944:
945:   for (std::size_t i = 0; i < hp.M[0]; ++i) {
946:     grid[0][i].min = i * M_STEP + hp.M[1];
947:     grid[0][i].max = grid[0][i].min + M_STEP;
948:   }
949:   for (std::size_t j = 0; j < hp.PT[0]; ++j) {
950:     grid[1][j].min = j * PT_STEP + hp.PT[1];
951:     grid[1][j].max = grid[1][j].min + PT_STEP;
952:   }
953:   for (std::size_t k = 0; k < hp.Y[0]; ++k) {
954:     grid[2][k].min = k * Y_STEP + hp.Y[1];
955:     grid[2][k].max = grid[2][k].min + Y_STEP;
956:   }
957:
958:   // ------------------------------------------------------------
959:   // Expand the detector transfer function
960:
961:   for (const auto &i : indices(grid[0])) {
962:     for (const auto &j : indices(grid[1])) {
963:       for (const auto &k : indices(grid[2])) {
964:         const std::vector<std::size_t> MC_ind = gra::spherical::GetIndices(
965:             MC, {grid[0][i].min, grid[0][i].max}, {grid[1][j].min, grid[1][j].max},
966:             {grid[2][k].min, grid[2][k].max});
967:
968:         // Acceptance mixing matrices
969:         fla_DET((i, j, k)).MIXlm = gra::spherical::GetGMixing(MC, MC_ind, param.LMAX, "fla");
970:         fid_DET((i, j, k)).MIXlm = gra::spherical::GetGMixing(MC, MC_ind, param.LMAX, "fid");
971:         det_DET((i, j, k)).MIXlm = gra::spherical::GetGMixing(MC, MC_ind, param.LMAX, "det");
972:
973:         // Get efficiency decomposition for this interval
974:         std::pair<std::vector<double>, std::vector<double>> E0 =
975:             gra::spherical::GetELM(MC, MC_ind, param.LMAX, "fla");
976:         std::pair<std::vector<double>, std::vector<double>> E1 =
977:             gra::spherical::GetELM(MC, MC_ind, param.LMAX, "fid");
978:         std::pair<std::vector<double>, std::vector<double>> E2 =
979:             gra::spherical::GetELM(MC, MC_ind, param.LMAX, "det");
980:
981:         fla_DET((i, j, k)).E_lm       = E0.first;
982:         fla_DET((i, j, k)).E_lm_error = E0.second;
983:
984:         fid_DET((i, j, k)).E_lm       = E1.first;
985:         fid_DET((i, j, k)).E_lm_error = E1.second;
986:
987:         det_DET((i, j, k)).E_lm       = E2.first;
988:         det_DET((i, j, k)).E_lm_error = E2.second;
989:       }
990:     }
991:   }
992:
993:   // ------------------------------------------------------------
994:
995:   // Loop over data sources
996:   for (const auto &ind : indices(DATA)) {
```



```
997:     // ------------------------------------------------------
998:     // Pre-Calculate once Spherical Harmonics for the MINUIT fit
999:     DATA_events = DATA[ind].EVENTS;
1000:    Y_lm        = gra::spherical::YLM(DATA_events, param.LMAX);
1001:    // ------------------------------------------------------
1002:
1003:    double chi2 = 0.0;
1004:
1005:    // Data source identifier string
1006:    const gra::spherical::Meta META = DATA[ind].META;
1007:
1008:    // Initialize tensor arrays
1009:    MTensor<gra::spherical::SH> temp(
1010:        {(unsigned int)hp.M[0], (unsigned int)hp.PT[0], (unsigned int)hp.Y[0]});
1011:    fla[META] = temp;
1012:    fid[META] = temp;
1013:    det[META] = temp;
1014:
1015:    // Expand Data
1016:    for (const auto &i : indices(grid[0])) {
1017:      for (const auto &j : indices(grid[1])) {
1018:        for (const auto &k : indices(grid[2])) {
1019:          // Data indices
1020:          DATA_ind = gra::spherical::GetIndices(DATA_events, {grid[0][i].min, grid[0][i].max},
1021:                                                {grid[1][j].min, grid[1][j].max},
1022:                                                {grid[2][k].min, grid[2][k].max});
1023:
1024:          const unsigned int MINEVENTS = 75;
1025:          if (DATA_ind.size() < MINEVENTS) {
1026:            std::cout << rang::fg::red << "WARNING: Less than " << MINEVENTS << " in the cell!"
1027:                      << rang::fg::reset << std::endl;
1028:          }
1029:
1030:          // ============================================================
1031:          // ALGORITHM 1: DIRECT / OBSERVED / ALGEBRAIC decomposition
1032:
1033:          fid[META]((i, j, k)).t_lm_MPP =
1034:              gra::spherical::SphericalMoments(DATA_events, DATA_ind, param.LMAX, "fid");
1035:          det[META]((i, j, k)).t_lm_MPP =
1036:              gra::spherical::SphericalMoments(DATA_events, DATA_ind, param.LMAX, "det");
1037:
1038:          // Forward matrix
1039:          Eigen::MatrixXd M = gra::aux::Matrix2Eigen(det_DET((i, j, k)).MIXlm);
1040:
1041:          // Matrix pseudoinverse
1042:          Eigen::VectorXd b = gra::aux::Vector2Eigen(det[META]((i, j, k)).t_lm_MPP);
1043:          Eigen::VectorXd x = gra::math::PseudoInverse(M, param.SVDREG) * b;
1044:
1045:          // Collect the inversion result
1046:          fla[META]((i, j, k)).t_lm_MPP = gra::aux::Eigen2Vector(x);
1047:
1048:          // ============================================================
1049:
1050:          // Update these to proper errors (TBD. PUT NAIVE POISSON
1051:          // COUNTING FOR NOW)
1052:          fla[META]((i, j, k)).t_lm_MPP_error =
1053:              std::vector<double>(fla[META]((i, j, k)).t_lm_MPP.size(), 0.0);
1054:          for (const auto &z : indices(fla[META]((i, j, k)).t_lm_MPP)) {
1055:            fla[META]((i, j, k)).t_lm_MPP_error[z] =
1056:                msqrt(std::abs(fla[META]((i, j, k)).t_lm_MPP[z]));
1057:          }
1058:
1059:          // Propagate errors assuming no error on mixing matrix
1060:          // (infinite reference MC statistics limit)
1061:          fid[META]((i, j, k)).t_lm_MPP_error =
1062:              spherical::ErrorProp(fid_DET((i, j, k)).MIXlm, fla[META]((i, j, k)).t_lm_MPP_error);
1063:          det[META]((i, j, k)).t_lm_MPP_error =
1064:              spherical::ErrorProp(det_DET((i, j, k)).MIXlm, fla[META]((i, j, k)).t_lm_MPP_error);
1065:
1066:          std::cout << "Algebraic Moore-Penrose/SVD inverted "
1067:                       "(unmixed) moments in the (angular flat) "
1068:                       "reference phase space:"
1069:                    << std::endl;
1070:          gra::spherical::PrintOutMoments(fla[META]((i, j, k)).t_lm_MPP,
1071:                                          fla[META]((i, j, k)).t_lm_MPP_error, ACTIVE, param.LMAX);
1072:
1073:          std::cout << "Algebraic (mixed) moments in the fiducial "
1074:                       "phase space:"
1075:                    << std::endl;
1076:          gra::spherical::PrintOutMoments(fid[META]((i, j, k)).t_lm_MPP,
1077:                                          fid[META]((i, j, k)).t_lm_MPP_error, ACTIVE, param.LMAX);
1078:
1079:          std::cout << "Algebraic (mixed) moments in the detector space:" << std::endl;
```



```
1080:          gra::spherical::PrintOutMoments(det[META]((i, j, k)).t_lm_MPP,
1081:                                          det[META]((i, j, k)).t_lm_MPP_error, ACTIVE, param.LMAX);
1082:
1083:          // ============================================================
1084:          // ALGORITHM 2: Extended Maximum Likelihood Fit
1085:
1086:          if (param.EML) {
1087:            // *** Get TMinuit based fit decomposition ***
1088:            MomentFit(META, (i, j, k), fitfunc);
1089:
1090:            // Save result
1091:            fla[META]((i, j, k)).t_lm_EML = t_lm;
1092:            fid[META]((i, j, k)).t_lm_EML =
1093:                fid_DET((i, j, k)).MIXlm * t_lm;  // Moment forward rotation: Matrix *
1094:            // Vector
1095:            det[META]((i, j, k)).t_lm_EML =
1096:                det_DET((i, j, k)).MIXlm * t_lm;  // Moment forward rotation: Matrix *
1097:            // Vector
1098:
1099:            // Uncertainties
1100:            fla[META]((i, j, k)).t_lm_EML_error = t_lm_error;
1101:
1102:            // Propagate errors assuming no error on mixing
1103:            // matrix (infinite reference MC statistics limit)
1104:            fid[META]((i, j, k)).t_lm_EML_error =
1105:                spherical::ErrorProp(fid_DET((i, j, k)).MIXlm, t_lm_error);
1106:            det[META]((i, j, k)).t_lm_EML_error =
1107:                spherical::ErrorProp(det_DET((i, j, k)).MIXlm, t_lm_error);
1108:            // ============================================================
1109:
1110:            std::cout << "Extended Maximum-Likelihood inverse "
1111:                         "fitted (unmixed) moments in the "
1112:                         "(angular flat)  phase space:"
1113:                      << std::endl;
1114:            gra::spherical::PrintOutMoments(fla[META]((i, j, k)).t_lm_EML,
1115:                                            fla[META]((i, j, k)).t_lm_EML_error, ACTIVE,
1116:                                            param.LMAX);
1117:
1118:            std::cout << "Extended Maximum-Likelihood "
1119:                         "Re-Back-Projected (mixed) moments in "
1120:                         "the fiducial phase space:"
1121:                      << std::endl;
1122:            gra::spherical::PrintOutMoments(fid[META]((i, j, k)).t_lm_EML,
1123:                                            fid[META]((i, j, k)).t_lm_EML_error, ACTIVE,
1124:                                            param.LMAX);
1125:
1126:            std::cout << "Extended Maximum-Likelihood "
1127:                         "Re-Back-Projected (mixed) moments in "
1128:                         "the detector space:"
1129:                      << std::endl;
1130:            gra::spherical::PrintOutMoments(det[META]((i, j, k)).t_lm_EML,
1131:                                            det[META]((i, j, k)).t_lm_EML_error, ACTIVE,
1132:                                            param.LMAX);
1133:          }
1134:
1135:          // ------------------------------------------------------------
1136:          // Print results
1137:          chi2 += PrintOutHyperCell(META, (i, j, k));
1138:
1139:          // ------------------------------------------------------------
1140:          // Make comparison of synthetic MC data vs estimate
1141:
1142:          if (DATA[ind].META.MODE == "MC") {
1143:            int fiducial = 0;
1144:            int selected = 0;
1145:            for (const auto &i : DATA_ind) {
1146:              if (DATA_events[i].fiducial) { ++fiducial; }
1147:              if (DATA_events[i].fiducial && DATA_events[i].selected) { ++selected; }
1148:            }
1149:            std::cout << std::endl;
1150:
1151:            std::cout << rang::fg::yellow << "MC GROUND TRUTH: " << rang::fg::reset << std::endl;
1152:            printf(
1153:                "   MC 'synthetic data' events generated       "
1154:                "  = %u \n",
1155:                (unsigned int)DATA_ind.size());
1156:            printf(
1157:                "   MC 'synthetic data' events fiducial        "
1158:                "  = %d (acceptance %0.1f percent) \n",
1159:                fiducial, fiducial / (double)DATA_ind.size() * 100);
1160:            printf(
1161:                "   MC 'synthetic data' events fiducial and "
1162:                "selected = %d (efficiency %0.1f percent) \n",
```



```
1163:                selected, selected / (double)fiducial * 100);
1164:            std::cout << std::endl;
1165:          }
1166:        }
1167:      }
1168:    }
1169:
1170:    if (param.EML) {
1171:      gra::aux::PrintBar("=");
1172:      double reducedchi2 = chi2 / (double)(ACTIVENDF * hp.M[0] * hp.PT[0] * hp.Y[0]);
1173:      if (reducedchi2 < 3) {
1174:        std::cout << rang::fg::green;
1175:      } else {
1176:        std::cout << rang::fg::red;
1177:      }
1178:      printf(
1179:          "Total chi2(MPP - EML) / (ACTIVENDF x BINS) = %0.2f / (%d x %d) = "
1180:          "%0.2F \n",
1181:          chi2, ACTIVENDF, (int)(hp.M[0] * hp.PT[0] * hp.Y[0]), reducedchi2);
1182:
1183:      std::cout << rang::fg::reset;
1184:      gra::aux::PrintBar("=");
1185:      std::cout << std::endl << std::endl;
1186:    }
1187:
1188:  }  // Loop over data sources
1189: }
1190:
1191: // Print out results for the hypercell
1192: //
1193: double MHarmonic::PrintOutHyperCell(const gra::spherical::Meta &    META,
1194:                                     const std::vector<std::size_t> &cell) {
1195:   double chi2 = 0;
1196:
1197:   // Print information
1198:   META.Print();
1199:
1200:   // Extended Maximum Likelihood
1201:   if (param.EML == true) {
1202:     // Loop over moments
1203:     for (int l = 0; l <= param.LMAX; ++l) {
1204:       for (int m = -l; m <= l; ++m) {
1205:         const int index = gra::spherical::LinearInd(l, m);
1206:
1207:         const double obs = det[META](cell).t_lm_MPP[index];
1208:         const double fit = det[META](cell).t_lm_EML[index];
1209:
1210:         // Moment active
1211:         if (ACTIVE[index]) { chi2 += pow2(obs - fit) / pow2(obs); }
1212:       }
1213:     }
1214:     std::cout << std::endl;
1215:
1216:     const double reducedchi2 = chi2 / (double)ACTIVENDF;
1217:     if (reducedchi2 < 3) {
1218:       std::cout << rang::fg::green;
1219:     } else {
1220:       std::cout << rang::fg::red;
1221:     }
1222:     printf("chi2(MPP - EML) / ndf = %0.3f / %d = %0.3f \n", chi2, ACTIVENDF, reducedchi2);
1223:     std::cout << rang::fg::reset << std::endl;
1224:
1225:     const double sum_fla = fla[META](cell).t_lm_EML[gra::spherical::LinearInd(0, 0)];
1226:     printf(
1227:         "EML: Estimate of events in this hyperbin in the flat phase space = "
1228:         "%0.1f +- %0.1f "
1229:         "\n",
1230:         sum_fla, msqrt(sum_fla));  // Poisson error
1231:
1232:     const double sum_FID = gra::spherical::HarmDotProd(fid_DET(cell).E_lm, fla[META](cell).t_lm_EML,
1233:                                                        ACTIVE, param.LMAX);
1234:     printf(
1235:         "EML: Estimate of events in this hyperbin in the fiducial  phase "
1236:         "space = %0.1f "
1237:         "+- %0.1f \n",
1238:         sum_FID, msqrt(sum_FID));  // Poisson error
1239:
1240:     const double sum_DET = gra::spherical::HarmDotProd(det_DET(cell).E_lm, fla[META](cell).t_lm_EML,
1241:                                                        ACTIVE, param.LMAX);
1242:     printf(
1243:         "EML: Estimate of events in this hyperbin in the detector        "
1244:         "space = %0.1f "
1245:         "+- %0.1f \n",
```

```
1246:         sum_DET, msqrt(sum_DET));  // Poisson error
1247:   }
1248:   std::cout << std::endl;
1249:
1250:   // Algebraic inverse
1251:   const double sum_fla = fla[META](cell).t_lm_MPP[gra::spherical::LinearInd(0, 0)];
1252:   printf(
1253:       "MPP: Estimate of events in this hyperbin in the flat phase space = %0.1f "
1254:       "+- %0.1f \n",
1255:       sum_fla, msqrt(sum_fla));  // Poisson error
1256:
1257:   const double sum_FID =
1258:       gra::spherical::HarmDotProd(fid_DET(cell).E_lm, fla[META](cell).t_lm_MPP, ACTIVE, param.LMAX);
1259:   printf(
1260:       "MPP: Estimate of events in this hyperbin in the fiducial  phase "
1261:       "space = %0.1f +- "
1262:       "%0.1f \n",
1263:       sum_FID, msqrt(sum_FID));  // Poisson error
1264:
1265:   const double sum_DET =
1266:       gra::spherical::HarmDotProd(det_DET(cell).E_lm, fla[META](cell).t_lm_MPP, ACTIVE, param.LMAX);
1267:   printf(
1268:       "MPP: Estimate of events in this hyperbin in the detector        "
1269:       "space = %0.1f +- "
1270:       "%0.1f \n",
1271:       sum_DET, msqrt(sum_DET));  // Poisson error
1272:
1273:   return chi2;
1274: }
1275:
1276: // MINUIT based fit routine for the Extended Maximum Likelihood formalism
1277: //
1278: void MHarmonic::MomentFit(const gra::spherical::Meta &META, const std::vector<std::size_t> &cell,
1279:                           void (*fitfunc)(int &, double *, double &, double *, int)) {
1280:   std::cout << "MomentFit: Starting ..." << std::endl << std::endl;
1281:
1282:   // **** This must be set for the loss function ****
1283:   activecell = cell;
1284:
1285:   // Init TMinuit
1286:   TMinuit *gMinuit = new TMinuit(NCOEF);  // initialize TMinuit with a maximum of N params
1287:   gMinuit->SetFCN(fitfunc);
1288:
1289:   // Set Print Level
1290:   // -1 no output
1291:   // 1 standard output
1292:   gMinuit->SetPrintLevel(-1);
1293:
1294:   // Set error Definition
1295:   // 1 for Chi square
1296:   // 0.5 for negative log likelihood
1297:   //   gMinuit->SetErrorDef(0.5);
1298:   double arglist[2];
1299:   int    ierflg = 0;
1300:   arglist[0]    = 0.5;  // 0.5 <=> We use negative log likelihood cost function
1301:   gMinuit->mnexcm("SET ERR", arglist, 1, ierflg);
1302:
1303:   // double fminbest = 1e32;
1304:   // Try different initial values to find out true minimum
1305:   const std::size_t TRIALMAX = 1;
1306:
1307:   for (std::size_t trials = 0; trials < TRIALMAX; ++trials) {
1308:     for (int l = 0; l <= param.LMAX; ++l) {
1309:       for (int m = -l; m <= l; ++m) {
1310:         const int index = gra::spherical::LinearInd(l, m);
1311:
1312:         const std::string str = "t_" + std::to_string(l) + std::to_string(m);
1313:
1314:         const double max = 1e9;  // Physically bounded ultimately by the number of events
1315:         const double min = -1e5;
1316:
1317:         // ========================================================
1318:         // *** Use the algebraic inverse solution as a good starting value
1319:         // ***
1320:         const double start_value = fla[META](activecell).t_lm_MPP[index];
1321:         // ========================================================
1322:
1323:         const double step_value = 0.1;  // in units of Events
1324:
1325:         gMinuit->mnparm(index, str, start_value, step_value, min, max, ierflg);
1326:
1327:         // After first trial, fix t_00 (error are not estimated with
1328:         // constant parameters)
```



```
1329:         // if (trials > 0) {
1330:         //      gMinuit->mnparm(0, "t_00", t_lm[0], 0, 0, 0, ierflg);
1331:         //      gMinuit->FixParameter(0);
1332:         //}
1333:         // FIX ODD MOMENTS TO ZERO
1334:         if (param.REMOVEODD && ((l % 2) != 0)) {
1335:           gMinuit->mnparm(index, str, 0, 0, 0, 0, ierflg);
1336:           gMinuit->FixParameter(index);
1337:         }
1338:         // FIX NEGATIVE M TO ZERO
1339:         if (param.REMOVENEGATIVEM && (m < 0)) {
1340:           gMinuit->mnparm(index, str, 0, 0, 0, 0, ierflg);
1341:           gMinuit->FixParameter(index);
1342:         }
1343:       }
1344:     }
1345:     // Scan main parameter
1346:     // gMinuit->mnscan();
1347:     arglist[0] = 100000;   // Minimum number of function calls
1348:     arglist[1] = 0.00001;  // Minimum tolerance
1349:
1350:     // First simplex to find approximate answers
1351:     gMinuit->mnexcm("SIMPLEX", arglist, 2, ierflg);
1352:
1353:     // Numerical Hessian (2nd derivatives matrix), inverse of this -> covariance
1354:     // matrix
1355:     gMinuit->mnexcm("MIGRAD", arglist, 2, ierflg);
1356:
1357:     // Confidence intervals based on the profile likelihood ratio
1358:     gMinuit->mnexcm("MINOS", arglist, 2, ierflg);
1359:
1360:     // Calculate error & covariance matrix
1361:     gMinuit->mnemat(&errmat[0][0], NCOEF);
1362:
1363:     for (int i = 0; i < NCOEF; ++i) {
1364:       for (int j = 0; j < NCOEF; ++j) {
1365:         covmat[i][j] = sqrt(errmat[i][i] * errmat[j][j]);
1366:         if (covmat[i][j] > 1E-80) {
1367:           covmat[i][j] = errmat[i][j] / covmat[i][j];
1368:         } else
1369:           covmat[i][j] = 0.0;
1370:       }
1371:     }
1372:     errmat.Print("Error matrix");
1373:     covmat.Print("Covariance matrix");
1374:
1375:     // Print results
1376:     double fmin, fedm, errdef  = 0.0;
1377:     int    nvpar, nparx, istat = 0;
1378:     gMinuit->mnstat(fmin, fedm, errdef, nvpar, nparx, istat);
1379:
1380:     // Collect fit result
1381:     for (int l = 0; l <= param.LMAX; ++l) {
1382:       for (int m = -l; m <= l; ++m) {
1383:         const int index = gra::spherical::LinearInd(l, m);
1384:
1385:         // Set results into t_lm[], t_lm_error[]
1386:         gMinuit->GetParameter(index, t_lm[index], t_lm_error[index]);
1387:       }
1388:     }
1389:
1390:   }  // TRIALS LOOP END
1391:
1392:   // Sum of N random variables Y = X_1 + X_2 + ... X_N
1393:   // sigma_Y^2 = \sum_i^N \sigma_i^2 + 2 \sum_i \sum_i < j
1394:   // covariance(X_i, X_j)
1395:
1396:   // Print out results (see MINUIT manual for these parameters)
1397:   double fmin, fedm, errdef  = 0.0;
1398:   int    nvpar, nparx, istat = 0;
1399:   gMinuit->mnstat(fmin, fedm, errdef, nvpar, nparx, istat);
1400:   gMinuit->mnprin(4, fmin);
1401:
1402:   std::cout << "<MINUIT (MIGRAD+MINOS)> done." << std::endl << std::endl;
1403:
1404:   delete gMinuit;
1405: }
1406:
1407: // Unbinned Extended Maximum Likelihood function
1408: // Extended means that the number of events (itself) is a Poisson distributed
1409: // random variable and that is incorporated to the fit.
1410: void MHarmonic::logLfunc(int &npar, double *gin, double &f, double *par, int iflag) const {
1411:   // Collect fit t_LM coefficients from MINUIT
```



```
1412:   std::vector<double> T(NCOEF, 0.0);
1413:
1414:   for (const auto &i : indices(T)) {
1415:     T[i] = par[i];
1416:     // printf("T[%d] = %0.5f \n", i);
1417:   }
1418:   // This is the number of events in the current phase space point at detector
1419:   // level
1420:   const int METHOD = 2;
1421:   double    nhat   = 0.0;
1422:
1423:   // Equivalent estimator 1
1424:   if (METHOD == 1) {
1425:     const std::vector<double> t_lm_det = det_DET(activecell).MIXlm * T;  // Matrix * Vector
1426:     nhat                               = t_lm_det[0];
1427:   }
1428:   // Equivalent estimator 2
1429:   if (METHOD == 2) {
1430:     nhat = gra::spherical::HarmDotProd(det_DET(activecell).E_lm, T, ACTIVE, param.LMAX);
1431:   }
1432:
1433:   // For each event, calculate \sum_{LM} t_{lm}
1434:   // Re[Y_{lm}(costheta,phi_k)], k is the event index
1435:   std::vector<double> I0;
1436:   const double        V = msqrt(4.0 * PI);  // Normalization volume
1437:
1438:   for (const auto &k : DATA_ind) {
1439:     // Event is accepted
1440:     if (DATA_events[k].fiducial && DATA_events[k].selected) {
1441:       // fine
1442:     } else {
1443:       continue;
1444:     }
1445:
1446:     // Loop over (l,m) terms
1447:     double sum = 0.0;
1448:     for (int l = 0; l <= param.LMAX; ++l) {
1449:       for (int m = -l; m <= l; ++m) {
1450:         const int index = gra::spherical::LinearInd(l, m);
1451:
1452:         if (ACTIVE[index]) {
1453:           // Calculate here
1454:           // const std::complex<double> Y =
1455:           //     gra::math::Y_complex_basis(DATA_events[k].costheta,DATA_events[k].phi,
1456:           // l,m);
1457:           // const double ReY = gra::math::NReY(Y,l,m);
1458:
1459:           // Pre-calculated for speed
1460:           const double ReY = Y_lm[k][index];
1461:
1462:           // Add
1463:           sum += T[index] * ReY;
1464:         }
1465:       }
1466:     }
1467:     I0.push_back(V * sum);
1468:   }
1469:   // Fidelity term
1470:   double logL = 0;
1471:
1472:   for (const auto &k : indices(I0)) {
1473:     if (I0[k] > 0) {
1474:       logL += std::log(I0[k]);
1475:       // Positivity is not satisfied, make strong penalty
1476:     } else {
1477:       logL = -1e32;
1478:       // The Re[Y_lm] (purely real) formulation here does
1479:       // not, explicitly, enforce non-negativity.
1480:       // Where as |A|^2 formulation would enforce it by
1481:       // construction,
1482:       // but that would have -l <= m <= l parameters, i.e.,
1483:       // over redundant representation problem
1484:       // on the other hand.
1485:     }
1486:   }
1487:
1488:   // Note the minus sign on logL term
1489:   f = -logL + nhat;
1490:
1491:   // High penalty, total event count estimate has gone out of physical
1492:   if (nhat < 0) { f = 1e32; }
1493:
1494:   // L1-norm regularization (Laplace prior), -> + ln(P_laplace)
```



```
1495:   double             llterm = 0.0;
1496:   const unsigned int START  = 1;
1497:   for (std::size_t i = START; i < T.size(); ++i) { llterm += std::abs(T[i]); }
1498:   f += llterm * param.LlREG * nhat;  // nhat for scale normalization
1499:
1500:   // printf("MHarmonic:: cost-functional = %0.4E, nhat = %0.1f \n", f, nhat);
1501: }
1502:
1503: }  // namespace gra
1504:
```

```cpp
// Fast MC analysis class
//
// (c) 2017-2020 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

// C++
#include <algorithm>
#include <iostream>
#include <memory>
#include <vector>

// C-file processing
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

// ROOT
#include "TBranch.h"
#include "TCanvas.h"
#include "TColor.h"
#include "TF1.h"
#include "TFile.h"
#include "TH1.h"
#include "TH2.h"
#include "TLegend.h"
#include "TLine.h"
#include "TProfile.h"
#include "TROOT.h"
#include "TRandom3.h"
#include "TString.h"
#include "TStyle.h"
#include "TTree.h"

// HepMC3 3
#include "HepMC3/FourVector.h"
#include "HepMC3/GenCrossSection.h"
#include "HepMC3/GenEvent.h"
#include "HepMC3/GenParticle.h"
#include "HepMC3/GenVertex.h"
#include "HepMC3/Print.h"
#include "HepMC3/ReaderAscii.h"
#include "HepMC3/Relatives.h"
#include "HepMC3/Selector.h"

// Own
#include "Granitti/Analysis/MAnalyzer.h"
#include "Granitti/Analysis/MMultiplet.h"
#include "Granitti/M4Vec.h"
#include "Granitti/MKinematics.h"
#include "Granitti/MMath.h"
#include "Granitti/MPDG.h"

const bool DEBUG = false;

using gra::aux::indices;
using gra::math::msqrt;

namespace gra {

// Constructor with unique ID string for ROOT bookkeeping reasons
MAnalyzer::MAnalyzer(const std::string &ID) {
  // Initialize histograms
  const int NBINS = 150;

  // Energy
  hE_Pions        = std::make_shared<TH1D>(Form("%s_%s", "Energy #pi (GeV)", ID.c_str()),
                                           ";Energy (GeV);Events", NBINS, 0, sqrts / 2.0);
  hE_Gamma        = std::make_shared<TH1D>(Form("%s_%s", "Energy #gamma (GeV)", ID.c_str()),
                                           ";Energy (GeV);Events", NBINS, 0, sqrts / 2.0);
  hE_Neutron      = std::make_shared<TH1D>(Form("%s_%s", "Energy n (GeV)", ID.c_str()),
                                           ";Energy (GeV);Events", NBINS, 0, sqrts / 2.0);
  hE_GammaNeutron = std::make_shared<TH1D>(Form("%s_%s", "Energy y+n (GeV)", ID.c_str()),
                                           ";Energy (GeV);Events", NBINS, 0, sqrts / 2.0);

  // Feynman-x
  hXF_Pions   = std::make_shared<TH1D>(Form("%s_%s", "xF #pi", ID.c_str()), ";Feynman-x;Events",
                                       NBINS, -1.0, 1.0);
  hXF_Gamma   = std::make_shared<TH1D>(Form("%s_%s", "xF #gamma", ID.c_str()), ";Feynman-x;Events",
                                       NBINS, -1.0, 1.0);
  hXF_Neutron = std::make_shared<TH1D>(Form("%s_%s", "xF n", ID.c_str()), ";Feynman-x;Events",
                                       NBINS, -1.0, 1.0);

  // Forward systems
```



```cpp
  hEta_Pions =
      std::make_shared<TH1D>(Form("%s_%s", "#eta pi", ID.c_str()), ";#eta;Events", NBINS, -12, 12);
  hEta_Gamma =
      std::make_shared<TH1D>(Form("%s_%s", "#eta y", ID.c_str()), ";#eta;Events", NBINS, -12, 12);
  hEta_Neutron =
      std::make_shared<TH1D>(Form("%s_%s", "#eta n", ID.c_str()), ";#eta;Events", NBINS, -12, 12);
  hM_NSTAR =
      std::make_shared<TH1D>(Form("%s_%s", "M (GeV)", ID.c_str()), ";M (GeV);Events", NBINS, 0, 10);

  // Legendre polynomials, DO NOT CHANGE THE Y-RANGE [-1,1]
  for (std::size_t i = 0; i < 8; ++i) {
    hPl[i] =
        std::make_shared<TProfile>(Form("hPl%lu_%s", i + 1, ID.c_str()), "", 100, 0.0, 4.0, -1, 1);
    hPl[i]->Sumw2();  // Error saving on
    hPl[i]->SetXTitle(Form("System M (GeV)"));
    hPl[i]->SetYTitle(Form("Legendre #LTP_{l}(cos #theta)#GT [CM frame]"));
  }

  // Costheta correlations between different frames
  for (std::size_t i = 0; i < analyzer::FRAMES.size(); ++i) {
    for (std::size_t j = 0; j < analyzer::FRAMES.size(); ++j) {
      h2CosTheta[i][j] = std::make_shared<TH2D>(
          Form("%s^{+} cos(theta) %s vs %s_%s", pstr.c_str(), analyzer::FRAMES[i].c_str(),
               analyzer::FRAMES[j].c_str(), ID.c_str()),
          Form(";%s^{+} cos(#theta) %s;%s^{+} cos(#theta) %s", pstr.c_str(),
               analyzer::FRAMES[i].c_str(), pstr.c_str(), analyzer::FRAMES[j].c_str()),
          NBINS, -1, 1, NBINS, -1, 1);
    }
  }

  // Phi correlations between different frames
  for (std::size_t i = 0; i < analyzer::FRAMES.size(); ++i) {
    for (std::size_t j = 0; j < analyzer::FRAMES.size(); ++j) {
      h2Phi[i][j] = std::make_shared<TH2D>(
          Form("%s^{+} #phi %s vs %s_%s", pstr.c_str(), analyzer::FRAMES[i].c_str(),
               analyzer::FRAMES[j].c_str(), ID.c_str()),
          Form(";%s^{+} #phi %s (rad);%s^{+} #phi %s (rad)", pstr.c_str(),
               analyzer::FRAMES[i].c_str(), pstr.c_str(), analyzer::FRAMES[j].c_str()),
          NBINS, -gra::math::PI, gra::math::PI, NBINS, -gra::math::PI, gra::math::PI);
    }
  }
}

// Destructor
MAnalyzer::~MAnalyzer() {}

// "Oracle" histogram filler:
//
// Oracle here means that in this function we (may) use event tree information,
// not just pure fiducial final state information based on purely physical observables.
//
double MAnalyzer::HepMC3_OracleFill(const std::string input, unsigned int multiplicity,
                                    int finalPDG, unsigned int MAXEVENTS,
                                    std::map<std::string, std::shared_ptr<h1Multiplet>> &    h1,
                                    std::map<std::string, std::shared_ptr<h2Multiplet>> &    h2,
                                    std::map<std::string, std::shared_ptr<hProfMultiplet>> &hP,
                                    unsigned int                                            SID) {
  inputfile                 = input;
  const std::string   totalpath = gra::aux::GetBasePath(2) + "/output/" + input + ".hepmc3";
  HepMC3::ReaderAscii input_file(totalpath);

  if (input_file.failed()) {
    throw std::invalid_argument("MAnalyzer::HepMC3Read: Cannot open file " + totalpath);
  }

  // Event loop
  unsigned int events_read = 0;

  // Variables for calculating selection efficiency
  double totalW = 0;
  double selecW = 0;

  // ------------------------------------------------------------------
  // Set final state [charged pair or neutral pair]
  MPDG PDG;
  PDG.ReadParticleData(gra::aux::GetBasePath(2) + "/modeldata/mass_width_2018.mcd");

  // Try to find the particle from PDG table, will throw exception if fails
  MParticle p          = PDG.FindByPDG(finalPDG);
  const int NEGfinalPDG = (p.charge3 != 0) ? -finalPDG : 0;  // Do not double count neutral
  // ------------------------------------------------------------------

  while (true) {
```



```cpp
    // Read event from input file
    HepMC3::GenEvent evt(HepMC3::Units::GEV, HepMC3::Units::MM);
    input_file.read_event(evt);

    // Reading failed
    if (input_file.failed()) {
      if (events_read == 0) {
        throw std::invalid_argument("MAnalyzer::HepMC3Read: File " + totalpath + " is empty!");
      } else {
        break;
      }
    }
    if (events_read == 0) {
      HepMC3::Print::listing(evt);
      HepMC3::Print::content(evt);
    }
    ++events_read;

    // *** Get generator cross section (in picobarns by HepMC3 convention) ***
    std::shared_ptr<HepMC3::GenCrossSection> cs =
        evt.attribute<HepMC3::GenCrossSection>("GenCrossSection");
    if (cs) {
      cross_section = 1E-12 * cs->xsec(0);  // turn into barns
    } else {
      std::cout << "Problem accessing 'GenCrossSection' attribute!" << std::endl;
    }
    // ------------------------------------------------------------------

    // *** Get event weight (always in barn units) ***
    double W = 1.0;
    if (evt.weights().size() != 0) {  // check do we have weights saved
      W = evt.weights()[0];           // take the first one
    }
    totalW += W;
    // ------------------------------------------------------------------

    // Central particles
    std::vector<M4Vec> pip;
    std::vector<M4Vec> pim;

    for (HepMC3::ConstGenParticlePtr p1 :
         HepMC3::applyFilter(HepMC3::StandardSelector::PDG_ID == finalPDG, evt.particles())) {
      M4Vec pvec = gra::aux::HepMC2M4Vec(p1->momentum());

      // Check that ancestor is a central system
      std::vector<HepMC3::ConstGenParticlePtr> results =
          HepMC3::applyFilter(*abs(HepMC3::StandardSelector::PDG_ID) == PDG::PDG_system,
                              HepMC3::Relatives::ANCESTORS(p1));
      if (results.size() != 0) { pip.push_back(pvec); }
    }
    for (HepMC3::ConstGenParticlePtr p1 :
         HepMC3::applyFilter(HepMC3::StandardSelector::PDG_ID == NEGfinalPDG, evt.particles())) {
      M4Vec pvec = gra::aux::HepMC2M4Vec(p1->momentum());

      // Check that ancestor is a central system
      std::vector<HepMC3::ConstGenParticlePtr> results = HepMC3::applyFilter(
          HepMC3::StandardSelector::PDG_ID == PDG::PDG_system, HepMC3::Relatives::ANCESTORS(p1));
      if (results.size() != 0) { pim.push_back(pvec); }
    }

    // CHECK CONDITION
    if (pip.size() + pim.size() != (unsigned int)multiplicity) {
      printf(
          "MAnalyzer::ReadHepMC3:: Multiplicity condition not filled +[%lu] "
          "-[%lu] %d! \n",
          pip.size(), pim.size(), multiplicity);
      continue;  // skip event
    }

    // ------------------------------------------------------------------
    // CENTRAL SYSTEM plots
    M4Vec system;
    for (const auto &x : pip) { system += x; }
    for (const auto &x : pim) { system += x; }

    std::vector<HepMC3::GenParticlePtr> beam_protons =
        HepMC3::applyFilter(HepMC3::StandardSelector::STATUS == PDG::PDG_BEAM &&
                                HepMC3::StandardSelector::PDG_ID == PDG::PDG_p,
                            evt.particles());

    std::vector<HepMC3::GenParticlePtr> final_protons =
        HepMC3::applyFilter(HepMC3::StandardSelector::STATUS == PDG::PDG_STABLE &&
                                HepMC3::StandardSelector::PDG_ID == PDG::PDG_p,
```



```cpp
                            evt.particles());

    M4Vec p_beam_plus;
    M4Vec p_beam_minus;
    M4Vec p_final_plus;
    M4Vec p_final_minus;

    // If we have full event, check energy-momentum
    if (beam_protons.size()) {
      // ============================================================
      sqrts = CheckEnergyMomentum(evt);
      // ============================================================
    }

    // Beam (initial state ) protons
    for (const HepMC3::GenParticlePtr &p1 : beam_protons) {
      M4Vec pvec = gra::aux::HepMC2M4Vec(p1->momentum());
      if (pvec.Rap() > 0) {
        p_beam_plus = pvec;
      } else {
        p_beam_minus = pvec;
      }
    }

    // Final state protons
    for (const HepMC3::GenParticlePtr &p1 : final_protons) {
      M4Vec pvec = gra::aux::HepMC2M4Vec(p1->momentum());

      // Check that ancestor is NOT excited forward system or central system
      std::vector<HepMC3::GenParticlePtr> results =
          HepMC3::applyFilter(HepMC3::StandardSelector::PDG_ID == PDG::PDG_NSTAR ||
                                  HepMC3::StandardSelector::PDG_ID == -PDG::PDG_NSTAR ||
                                  HepMC3::StandardSelector::PDG_ID == PDG::PDG_system,
                              HepMC3::Relatives::ANCESTORS(p1));

      if (results.size() == 0) {
        if (pvec.Rap() > 0) {
          p_final_plus = pvec;
        } else {
          p_final_minus = pvec;
        }
      }
    }

    // Observables for 2-body case only
    if (multiplicity == 2) {
      FrameObservables(W, evt, p_beam_plus, p_beam_minus, p_final_plus, p_final_minus, pip, pim);
    }

    // Observables for N stars
    NStarObservables(W, evt);

    // ************************************************************
    // SUPERPLOTTER >>
    try {
      M4Vec a = pip[0];
      M4Vec b;
      if (pim.size() != 0) {  // Charged pair
        b = pim[0];
      } else {  // Neutral pair
        b = pip[1];
      }

      const double M  = system.M();
      const double Pt = system.Pt();
      const double Y  = system.Rap();

      // 1D: System
      h1["h1_S_M"]->h[SID]->Fill(M, W);
      h1["h1_S_Pt"]->h[SID]->Fill(Pt, W);
      h1["h1_S_Pt2"]->h[SID]->Fill(math::pow2(Pt), W);
      h1["h1_S_Y"]->h[SID]->Fill(Y, W);
      hP["hP_S_M_Pt"]->h[SID]->Fill(M, Pt, W);

      // 1D: 1-Body
      h1["h1_1B_pt"]->h[SID]->Fill(a.Pt(), W);
      h1["h1_1B_eta"]->h[SID]->Fill(a.Eta(), W);

      // 1D: Forward proton pair
      double deltaphi_pp = -1.0;
      if (p_final_plus.M() > 0) {
        // Mandelstam -t_1,2
        const double t1 = -(p_beam_plus - p_final_plus).M2();
```

```
333:         // const double t2 = -(p_beam_minus - p_final_minus).M2();
334:
335:         // Deltaphi
336:         deltaphi_pp          = p_final_plus.DeltaPhiAbs(p_final_minus);
337:         M4Vec      pp_diff = p_final_plus - p_final_minus;
338:         const double pp_dpt  = pp_diff.Pt();
339:
340:         h1["h1_PP_dphi"]->h[SID]->Fill(deltaphi_pp, W);
341:         h1["h1_PP_t1"]->h[SID]->Fill(t1, W);
342:         h1["h1_PP_dpt"]->h[SID]->Fill(pp_dpt, W);
343:
344:         h2["h2_S_M_dphipp"]->h[SID]->Fill(M, deltaphi_pp, W);
345:         h2["h2_S_M_dpt"]->h[SID]->Fill(M, pp_dpt, W);
346:         h2["h2_S_M_t"]->h[SID]->Fill(M, std::abs(t1), W);
347:       }
348:
349:       // 2D
350:       h2["h2_S_M_Pt"]->h[SID]->Fill(M, Pt, W);
351:       h2["h2_S_M_pt"]->h[SID]->Fill(M, a.Pt(), W);
352:
353:       // 2-Body only
354:       if (multiplicity == 2) {
355:         hP["hP_2B_M_dphi"]->h[SID]->Fill(M, a.DeltaPhi(b), W);
356:         h1["h1_2B_acop"]->h[SID]->Fill(1.0 - a.DeltaPhi(b) / gra::math::PI, W);
357:         h1["h1_2B_diffrap"]->h[SID]->Fill(b.Rap() - a.Rap(), W);
358:         h2["h2_2B_M_dphi"]->h[SID]->Fill(M, a.DeltaPhi(b), W);
359:         h2["h2_2B_etal_eta2"]->h[SID]->Fill(a.Eta(), b.Eta(), W);
360:
361:
362:         // Frame transform
363:         const int   direction = 1;
364:         const M4Vec X         = a + b;
365:
366:         std::vector<M4Vec> CM = {a, b};
367:         gra::kinematics::CMframe(CM, X);
368:
369:         std::vector<M4Vec> HX = {a, b};
370:         gra::kinematics::HXframe(HX, X);
371:
372:         std::vector<M4Vec> CS = {a, b};
373:         gra::kinematics::CSframe(CS, X, p_beam_plus, p_beam_minus);
374:
375:         std::vector<M4Vec> GJ = {a, b};
376:         gra::kinematics::GJframe(GJ, X, direction, p_beam_plus - p_final_plus,
377:                                  p_beam_minus - p_final_minus);
378:
379:         std::vector<M4Vec> PG = {a, b};
380:         gra::kinematics::PGframe(PG, X, direction, p_beam_plus, p_beam_minus);
381:
382:
383:         h1["h1_costheta_CM"]->h[SID]->Fill(CM[0].CosTheta(), W);
384:         h1["h1_costheta_HX"]->h[SID]->Fill(HX[0].CosTheta(), W);
385:         h1["h1_costheta_CS"]->h[SID]->Fill(CS[0].CosTheta(), W);
386:         h1["h1_costheta_GJ"]->h[SID]->Fill(GJ[0].CosTheta(), W);
387:         h1["h1_costheta_PG"]->h[SID]->Fill(PG[0].CosTheta(), W);
388:         h1["h1_costheta_LAB"]->h[SID]->Fill(a.CosTheta(), W);
389:
390:
391:         h1["h1_phi_CM"]->h[SID]->Fill(CM[0].Phi(), W);
392:         h1["h1_phi_HX"]->h[SID]->Fill(HX[0].Phi(), W);
393:         h1["h1_phi_CS"]->h[SID]->Fill(CS[0].Phi(), W);
394:         h1["h1_phi_GJ"]->h[SID]->Fill(GJ[0].Phi(), W);
395:         h1["h1_phi_PG"]->h[SID]->Fill(PG[0].Phi(), W);
396:         h1["h1_phi_LAB"]->h[SID]->Fill(a.Phi(), W);
397:
398:
399:         h2["h2_2B_costheta_phi_CM"]->h[SID]->Fill(CM[0].CosTheta(), CM[0].Phi(), W);
400:         h2["h2_2B_costheta_phi_HX"]->h[SID]->Fill(HX[0].CosTheta(), HX[0].Phi(), W);
401:         h2["h2_2B_costheta_phi_CS"]->h[SID]->Fill(CS[0].CosTheta(), CS[0].Phi(), W);
402:         h2["h2_2B_costheta_phi_GJ"]->h[SID]->Fill(GJ[0].CosTheta(), GJ[0].Phi(), W);
403:         h2["h2_2B_costheta_phi_PG"]->h[SID]->Fill(PG[0].CosTheta(), PG[0].Phi(), W);
404:         h2["h2_2B_costheta_phi_LAB"]->h[SID]->Fill(a.CosTheta(), a.Phi(), W);
405:
406:
407:         h2["h2_2B_M_costheta_CM"]->h[SID]->Fill(M, CM[0].CosTheta(), W);
408:         h2["h2_2B_M_costheta_HX"]->h[SID]->Fill(M, HX[0].CosTheta(), W);
409:         h2["h2_2B_M_costheta_CS"]->h[SID]->Fill(M, CS[0].CosTheta(), W);
410:         h2["h2_2B_M_costheta_GJ"]->h[SID]->Fill(M, GJ[0].CosTheta(), W);
411:         h2["h2_2B_M_costheta_PG"]->h[SID]->Fill(M, PG[0].CosTheta(), W);
412:         h2["h2_2B_M_costheta_LAB"]->h[SID]->Fill(M, a.CosTheta(), W);
413:
414:
415:         h2["h2_2B_M_phi_CM"]->h[SID]->Fill(M, CM[0].Phi(), W);
```



```
416:         h2["h2_2B_M_phi_HX"]->h[SID]->Fill(M, HX[0].Phi(), W);
417:         h2["h2_2B_M_phi_CS"]->h[SID]->Fill(M, CS[0].Phi(), W);
418:         h2["h2_2B_M_phi_GJ"]->h[SID]->Fill(M, GJ[0].Phi(), W);
419:         h2["h2_2B_M_phi_PG"]->h[SID]->Fill(M, PG[0].Phi(), W);
420:         h2["h2_2B_M_phi_LAB"]->h[SID]->Fill(M, a.Phi(), W);
421:
422:
423:         // ------------------------------------------------------------------------
424:         hP["hP_S_M_PL2_CM"]->h[SID]->Fill(M, math::LegendrePl(2, CM[0].CosTheta()), W);
425:         hP["hP_S_M_PL4_CM"]->h[SID]->Fill(M, math::LegendrePl(4, CM[0].CosTheta()), W);
426:         h2["h2_2B_etal_eta2"]->h[SID]->Fill(a.Eta(), b.Eta(), W);
427:         // ------------------------------------------------------------------------
428:       }
429:
430:       // 4-Body only
431:       if (multiplicity == 4) {
432:         // ...
433:       }
434:     } catch (...) {
435:       throw std::invalid_argument("MAnalyzer::HepMC3Read: Problem filling histogram!");
436:     }
437:
438:     // << SUPERPLOTTER
439:     // ************************************************************
440:
441:     if (events_read >= MAXEVENTS) {
442:       std::cout << "MAnalyzer::HepMC3Read: Maximum event count " << MAXEVENTS << " reached!";
443:       break;  // Enough events
444:     }
445:
446:     if (events_read % 10000 == 0) {
447:       std::cout << std::endl << "Events processed: " << events_read << std::endl;
448:     }
449:     // [THIS AS LAST!] Sum selected event weights
450:     selecW += W;
451:   }
452:   std::cout << std::endl;
453:   std::cout << "MAnalyzer::HepMC3Read: Events processed in total: " << events_read << std::endl;
454:
455:   // Close HepMC3 file
456:   input_file.close();
457:
458:   if (selecW == 0.0) {
459:     throw std::invalid_argument("MAnalyzer::HepMC3Read:: Valid events in <" + totalpath + ">" +
460:                                 " == 0 out of " + std::to_string(events_read));
461:   }
462:   // Take into account extra fiducial cut efficiency here
463:   double efficiency = selecW / totalW;
464:   printf("MAnalyzer::HepMC3Read: Fiducial cut efficiency: %0.3f \n", efficiency);
465:   std::cout << std::endl;
466:
467:   return cross_section * efficiency;
468: }
469:
470: // Sanity check
471: double MAnalyzer::CheckEnergyMomentum(HepMC3::GenEvent &evt) const {
472:   std::vector<HepMC3::GenParticlePtr> all_init = HepMC3::applyFilter(
473:       HepMC3::StandardSelector::STATUS == PDG::PDG_BEAM, evt.particles());  // Beam
474:
475:   std::vector<HepMC3::GenParticlePtr> all_final = HepMC3::applyFilter(
476:       HepMC3::StandardSelector::STATUS == PDG::PDG_STABLE, evt.particles());  // Final state
477:
478:   M4Vec beam(0, 0, 0, 0);
479:   for (const HepMC3::GenParticlePtr &p1 : all_init) {
480:     beam += gra::aux::HepMC2M4Vec(p1->momentum());
481:   }
482:   M4Vec final(0, 0, 0, 0);
483:   for (const HepMC3::GenParticlePtr &p1 : all_final) {
484:     final += gra::aux::HepMC2M4Vec(p1->momentum());
485:   }
486:   if (!gra::math::CheckEMC(beam - final)) {
487:     gra::aux::PrintWarning();
488:     std::cout << rang::fg::red << "Energy-Momentum not conserved!" << rang::fg::reset << std::endl;
489:     (beam - final).Print();
490:     HepMC3::Print::listing(evt);
491:     HepMC3::Print::content(evt);
492:   }
493:   return beam.M();
494: }
495:
496: // 2-body angular observables
497: void MAnalyzer::FrameObservables(double W, HepMC3::GenEvent &evt, const M4Vec &p_beam_plus,
498:                                  const M4Vec &p_beam_minus, const M4Vec &p_final_plus,
```



```
499:                                  const M4Vec &p_final_minus, const std::vector<M4Vec> &pip,
500:                                  const std::vector<M4Vec> &pim) {
501:   // Find index
502:   const auto ind = [&](const std::string str) {
503:     for (const auto &i : indices(analyzer::FRAMES)) {
504:       if (analyzer::FRAMES[i] == str) { return i; }
505:     }
506:     throw std::invalid_argument("MAnalyzer::FrameObservables: Unknown Lorentz frame: " + str);
507:   };
508:
509:   std::vector<M4Vec> pf;
510:
511:   if (pip.size() != 0 && pim.size() != 0) {  // Charged pair
512:     pf = {pip[0], pim[0]};
513:   }
514:   if (pip.size() == 2 && pim.size() == 0) {  // Neutral pair
515:     pf = {pip[0], pip[1]};
516:   }
517:
518:   // ------------------------------------------------------------------
519:   // Lorentz frame transformations
520:
521:   // Make copies
522:   std::vector<std::vector<M4Vec>> pions;
523:   for (std::size_t i = 0; i < analyzer::FRAMES.size(); ++i) { pions.push_back(pf); }
524:
525:   // System
526:   const M4Vec X         = pf[0] + pf[1];
527:   const int   direction = 1;  // PG and GJ
528:
529:   gra::kinematics::CMframe(pions[ind("CM")], X);
530:   gra::kinematics::HXframe(pions[ind("HX")], X);
531:   gra::kinematics::CSframe(pions[ind("CS")], X, p_beam_plus, p_beam_minus);
532:   gra::kinematics::GJframe(pions[ind("GJ")], X, direction, p_beam_plus - p_final_plus,
533:                            p_beam_minus - p_final_minus);
534:   gra::kinematics::PGframe(pions[ind("PG")], X, direction, p_beam_plus, p_beam_minus);
535:   pions[ind("LAB")] = pions[ind("LAB")];  // already there, do nothing
536:
537:   // No forward protons -> set zero
538:   if (p_final_plus.M() < 0.5) { pions[ind("GJ")] = {M4Vec(0, 0, 0, 0), M4Vec(0, 0, 0, 0)}; }
539:
540:   // ------------------------------------------------------------------
541:   // FILL HISTOGRAMS -->
542:
543:   // Legendre polynomials P_l cos(theta)
544:   for (std::size_t l = 0; l < 8; ++l) {  // note l+1
545:     // Take first daughter [0]
546:     double value = gra::math::LegendrePl((l + 1), pions[ind("CM")][0].CosTheta());
547:     hPl[l]->Fill(X.M(), value, W);
548:   }
549:
550:   // FRAME correlations
551:   for (std::size_t i = 0; i < analyzer::FRAMES.size(); ++i) {
552:     for (std::size_t j = 0; j < analyzer::FRAMES.size(); ++j) {
553:       h2CosTheta[i][j]->Fill(pions[i][0].CosTheta(), pions[j][0].CosTheta(), W);
554:       h2Phi[i][j]->Fill(pions[i][0].Phi(), pions[j][0].Phi(), W);
555:     }
556:   }
557: }
558:
559: // Forward system observables
560: void MAnalyzer::NStarObservables(double W, HepMC3::GenEvent &evt) {
561:   // Excite forward system particles
562:   std::vector<HepMC3::GenParticlePtr> search_gammas =
563:       HepMC3::applyFilter(HepMC3::StandardSelector::PDG_ID == PDG::PDG_gamma, evt.particles());
564:
565:   std::vector<HepMC3::GenParticlePtr> search_neutrons =
566:       HepMC3::applyFilter(HepMC3::StandardSelector::PDG_ID == PDG::PDG_n, evt.particles());
567:
568:   std::vector<HepMC3::GenParticlePtr> search_pip =
569:       HepMC3::applyFilter(HepMC3::StandardSelector::PDG_ID == PDG::PDG_pip, evt.particles());
570:
571:   std::vector<HepMC3::GenParticlePtr> search_pim =
572:       HepMC3::applyFilter(HepMC3::StandardSelector::PDG_ID == PDG::PDG_pim, evt.particles());
573:
574:   std::vector<HepMC3::GenParticlePtr> search_nstar =
575:       HepMC3::applyFilter(HepMC3::StandardSelector::PDG_ID == PDG::PDG_NSTAR ||
576:                           HepMC3::StandardSelector::PDG_ID == -PDG::PDG_NSTAR,
577:                           evt.particles());
578:
579:   // Find out if we excited one or two protons
580:   bool excited_plus  = false;
```



```
581:   bool excited_minus = false;
582:   for (const HepMC3::GenParticlePtr &p1 : search_nstar) {
583:     M4Vec pvec = gra::aux::HepMC2M4Vec(p1->momentum());
584:     hM_NSTAR->Fill(pvec.M(), W);
585:
586:     if (pvec.Rap() > 0) { excited_plus = true; }
587:     if (pvec.Rap() < 0) { excited_minus = true; }
588:   }
589:   // Excited system found
590:   if (excited_plus || excited_minus) { N_STAR_ON = true; }
591:
592:   // N* system decay products
593:
594:   // Gammas
595:   double gamma_e_plus  = 0;
596:   double gamma_e_minus = 0;
597:
598:   // Gammas
599:   for (const HepMC3::GenParticlePtr &p1 : search_gammas) {
600:     M4Vec pvec = gra::aux::HepMC2M4Vec(p1->momentum());
601:
602:     // Check that ancestor is excited forward system
603:     std::vector<HepMC3::GenParticlePtr> ancestor =
604:         HepMC3::applyFilter(HepMC3::StandardSelector::PDG_ID == PDG::PDG_NSTAR ||
605:                             HepMC3::StandardSelector::PDG_ID == -PDG::PDG_NSTAR,
606:                             HepMC3::Relatives::ANCESTORS(p1));
607:
608:     if (ancestor.size() != 0) {
609:       hEta_Gamma->Fill(pvec.Eta(), W);
610:       hE_Gamma->Fill(pvec.E(), W);
611:       hXF_Gamma->Fill(pvec.Pz() / (sqrts / 2), W);
612:
613:       if (excited_plus && pvec.Rap() > 0) { gamma_e_plus += pvec.E(); }
614:       if (excited_minus && pvec.Rap() < 0) { gamma_e_minus += pvec.E(); }
615:     }
616:   }
617:
618:   // Pi+
619:   for (const HepMC3::GenParticlePtr &p1 : search_pip) {
620:     // HepMC3::Print::line(p1);
621:
622:     // Check that parent is the excited system
623:     std::vector<HepMC3::GenParticlePtr> parents = p1->parents();
624:     bool                                found   = false;
625:     for (const auto &k : indices(parents)) {
626:       if (std::abs(parents[k]->pid()) == PDG::PDG_NSTAR) { found = true; }
627:     }
628:     if (found) {
629:       M4Vec pvec = gra::aux::HepMC2M4Vec(p1->momentum());
630:       hEta_Pions->Fill(pvec.Eta(), W);
631:       hE_Pions->Fill(pvec.E(), W);
632:       hXF_Pions->Fill(pvec.Pz() / (sqrts / 2), W);
633:     }
634:   }
635:
636:   // Pi-
637:   for (const HepMC3::GenParticlePtr &p1 : search_pim) {
638:     // HepMC3::Print::line(p1);
639:
640:     // Check that parent is the excited system
641:     std::vector<HepMC3::GenParticlePtr> parents = p1->parents();
642:     bool                                found   = false;
643:     for (const auto &k : indices(parents)) {
644:       if (std::abs(parents[k]->pid()) == PDG::PDG_NSTAR) { found = true; }
645:     }
646:     if (found) {
647:       M4Vec pvec = gra::aux::HepMC2M4Vec(p1->momentum());
648:       hEta_Pions->Fill(pvec.Eta(), W);
649:       hE_Pions->Fill(pvec.E(), W);
650:       hXF_Pions->Fill(pvec.Pz() / (sqrts / 2), W);
651:     }
652:   }
653:
654:   // Neutrons
655:   double neutron_e_plus  = 0;
656:   double neutron_e_minus = 0;
657:
658:   for (const HepMC3::GenParticlePtr &p1 : search_neutrons) {
659:     M4Vec pvec = gra::aux::HepMC2M4Vec(p1->momentum());
660:
661:     // Check that parent is the excited system
662:     std::vector<HepMC3::GenParticlePtr> parents = p1->parents();
663:     bool                                found   = false;
664:
```

./src/Analysis/MAnalyzer.cc 9/11

```
665:     for (const auto &k : indices(parents)) {
666:       if (std::abs(parents[k]->pid()) == PDG::PDG_NSTAR) { found = true; }
667:     }
668:     if (found) {
669:       hEta_Neutron->Fill(pvec.Eta(), W);
670:       hE_Neutron->Fill(pvec.E(), W);
671:       hXF_Neutron->Fill(pvec.Pz() / (sqrts / 2), W);
672:
673:       if (excited_plus && pvec.Rap() > 0) { neutron_e_plus += pvec.E(); }
674:       if (excited_minus && pvec.Rap() < 0) { neutron_e_minus += pvec.E(); }
675:     }
676:   }
677:
678:   // Gamma+Neutron energy histogram
679:   if (excited_plus) { hE_GammaNeutron->Fill(gamma_e_plus + neutron_e_plus, W); }
680:   if (excited_minus) { hE_GammaNeutron->Fill(gamma_e_minus + neutron_e_minus, W); }
681: }
682:
683: double powerlaw(double *x, double *par) {
684:   return par[0] / std::pow(1.0 + std::pow(x[0], 2) / (std::pow(par[1], 2) * par[2]), par[2]);
685: }
686:
687: double exponential(double *x, double *par) { return par[0] * exp(par[1] * x[0]); }
688:
689: // Custom plotter
690: void MAnalyzer::PlotAll(const std::string &titlestr) {
691:   // Create output directory if it does not exist
692:   const std::string FOLDER = gra::aux::GetBasePath(2) + "/figs/" + inputfile;
693:   aux::CreateDirectory(FOLDER);
694:
695:   /*
696:   // FIT FUNCTIONS
697:   std::shared_ptr<TF1> fb = std::make_shared<TF1>("exp_fit", exponential, 0.05,
698:   0.5, 2);
699:   fb->SetParameter(0, 10.0); // A
700:   fb->SetParameter(1, -8.0); // b
701:
702:   std::shared_ptr<TF1> fa = std::make_shared<TF1>("pow_fit", powerlaw, 0.5, 3.0,
703:   3);
704:   fa->SetParameter(0, 10.0); // A
705:   fa->SetParameter(1, 0.15); // T
706:   fa->SetParameter(2, 0.1);  // n
707:   */
708:   //        hpT2->Fit("exp_fit","R");
709:   //        hpT_Meson_p->Fit("pow_fit","R"); // "R" for range
710:
711:   // *************** FORWARD EXCITATION ***************
712:   if (N_STAR_ON) {
713:     // Draw histograms
714:     TCanvas c1("c", "c", 800, 600);
715:     c1.Divide(2, 2, 0.0002, 0.0002);
716:
717:     c1.cd(1);
718:     // gPad->SetLogy();
719:     hEta_Gamma->SetLineColor(2);
720:     hEta_Pions->SetLineColor(4);
721:     hEta_Neutron->SetLineColor(8);
722:
723:     hEta_Gamma->Draw("same");
724:     hEta_Neutron->Draw("same");
725:     hEta_Pions->Draw("same");
726:
727:     // Set x-axis range
728:     const double eta_min = 3;
729:     const double eta_max = 15;
730:     hEta_Pions->SetAxisRange(eta_min, eta_max, "X");
731:     hEta_Gamma->SetAxisRange(eta_min, eta_max, "X");
732:     hEta_Neutron->SetAxisRange(eta_min, eta_max, "X");
733:
734:     c1.cd(2);
735:     // gPad->SetLogy();
736:     hXF_Gamma->SetLineColor(2);
737:     hXF_Pions->SetLineColor(4);
738:     hXF_Neutron->SetLineColor(8);
739:
740:     hXF_Gamma->Draw("same");
741:     hXF_Pions->Draw("same");
742:     hXF_Neutron->Draw("same");
743:
744:     c1.cd(3);
745:     gPad->SetLogy();
746:     // gPad->SetLogx();
747:     hM_NSTAR->Draw();
```

./src/Analysis/MAnalyzer.cc 10/11

```
748:
749:     c1.cd(4);
750:     gPad->SetLogy();
751:     hXF_Gamma->SetLineColor(2);
752:     hXF_Pions->SetLineColor(4);
753:     hXF_Neutron->SetLineColor(8);
754:
755:     hXF_Gamma->Draw("same");
756:     hXF_Pions->Draw("same");
757:     hXF_Neutron->Draw("same");
758:
759:     c1.SaveAs(Form("%s/figs/%s/forward.pdf", gra::aux::GetBasePath(2).c_str(), inputfile.c_str()));
760:   }
761:   // ************** *************** ***************
762:
763:   // ------------------------------------------------------------------------
764:   // FRAME correlations
765:
766:   TCanvas c2("c", "c", 800, 800);
767:   c2.Divide(analyzer::FRAMES.size(), analyzer::FRAMES.size(), 0.0001, 0.0002);
768:
769:   int k = 1;
770:   for (std::size_t i = 0; i < analyzer::FRAMES.size(); ++i) {
771:     for (std::size_t j = 0; j < analyzer::FRAMES.size(); ++j) {
772:       c2.cd(k);
773:       ++k;
774:       if (j >= i) { h2CosTheta[i][j]->Draw("COLZ"); }
775:
776:       // Titlestr
777:       if ((i == 0) & (j == 0)) { h2CosTheta[i][j]->SetTitle(titlestr.c_str()); }
778:     }
779:   }
780:   c2.SaveAs(Form("%s/figs/%s/h2_frame_correlations_costheta.pdf", gra::aux::GetBasePath(2).c_str(),
781:                  inputfile.c_str()));
782:
783:   // ------------------------------------------------------------------------
784:
785:   TCanvas c3("c", "c", 800, 800);
786:   c3.Divide(analyzer::FRAMES.size(), analyzer::FRAMES.size(), 0.0001, 0.0002);
787:
788:   k = 1;
789:   for (std::size_t i = 0; i < analyzer::FRAMES.size(); ++i) {
790:     for (std::size_t j = 0; j < analyzer::FRAMES.size(); ++j) {
791:       c3.cd(k);
792:       ++k;
793:       if (j >= i) { h2Phi[i][j]->Draw("COLZ"); }
794:
795:       // Titlestr
796:       if ((i == 0) & (j == 0)) { h2Phi[i][j]->SetTitle(titlestr.c_str()); }
797:     }
798:   }
799:   c3.SaveAs(Form("%s/figs/%s/h2_frame_correlations_phi.pdf", gra::aux::GetBasePath(2).c_str(),
800:                  inputfile.c_str()));
801:
802:   // ------------------------------------------------------------------------
803:   // ------------------------------------------------------------------------
804:   // Legendre polynomials in the Rest Frame (non-rotated one)
805:
806:   const int    colors[8] = {48, 53, 98, 32, 48, 53, 98, 32};
807:   const double XBOUND[2] = {0.0, 4.0};
808:
809:   // 1...8
810:   TCanvas *c115 = new TCanvas("c115", "Legendre polynomials 1-4", 600, 400);
811:   TCanvas *c116 = new TCanvas("c116", "Legendre polynomials 5-8", 600, 400);
812:
813:   TLegend *leg[8];
814:   TLine * line[8];
815:   c115->Divide(2, 2, 0.001, 0.001);
816:   c116->Divide(2, 2, 0.001, 0.001);
817:
818:   for (std::size_t l = 0; l < 8; ++l) {
819:     if (l < 4) {
820:       c115->cd(l + 1);
821:     } else {
822:       c116->cd(l - 3);
823:     }
824:
825:     // Legend
826:     leg[l] = new TLegend(0.15, 0.75, 0.4, 0.85);  // x1,y1,x2,y2
827:
828:     // Data
829:     hPl[l]->SetLineColor(colors[l]);
830:     hPl[l]->Draw("SAME");
```

./src/Analysis/MAnalyzer.cc 11/11

```
831:     hPl[l]->SetMinimum(-0.4);  // Y-axis minimum
832:     hPl[l]->SetMaximum(0.4);   // Y-axis maximum
833:
834:     // Adjust legend
835:     leg[l]->SetFillColor(0);   // White background
836:     leg[l]->SetBorderSize(0);  // No box
837:     leg[l]->AddEntry(hPl[l].get(), Form("l = %lu", l + 1), "l");
838:     leg[l]->Draw("SAME");
839:
840:     // Horizontal line
841:     line[l] = new TLine(XBOUND[0], 0, XBOUND[1], 0);
842:     line[l]->Draw("SAME");
843:
844:     // Titlestr
845:     if (l == 0 || l == 4) { hPl[l]->SetTitle(titlestr.c_str()); }
846:   }
847:   c115->SaveAs(
848:       Form("%s/figs/%s/hPl_1to4.pdf", gra::aux::GetBasePath(2).c_str(), inputfile.c_str()));
849:   c116->SaveAs(
850:       Form("%s/figs/%s/hPl_6to8.pdf", gra::aux::GetBasePath(2).c_str(), inputfile.c_str()));
851:
852:   for (std::size_t i = 0; i < 8; ++i) {
853:     delete leg[i];
854:     delete line[i];
855:   }
856:   delete c115;
857:   delete c116;
858: }
859:
860: }  // namespace gra
861:
```

./src/Analysis/MMultiplet.cc 1/8

```
1: // (Histo0, Histo1, Histo2, ..., Histo N-1) Multiplet ROOT histograms
2: //
3: // (c) 2017-2020 Mikael Mieskolainen
4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
5:
6: // C++
7: #include <string>
8: #include <tuple>
9:
10: // ROOT
11: #include "TColor.h"
12: #include "TGaxis.h"
13: #include "TH1.h"
14: #include "TH2.h"
15: #include "TLatex.h"
16: #include "TLegend.h"
17: #include "TPaveText.h"
18: #include "TProfile.h"
19:
20: // Own
21: #include "Graniitti/Analysis/MMultiplet.h"
22: #include "Graniitti/Analysis/MROOT.h"
23: #include "Graniitti/MAux.h"
24:
25: using gra::aux::indices;
26:
27: namespace gra {
28:
29: void GetLegendPosition(unsigned int N, double &x1, double &x2, double &y1, double &y2,
30:                        const std::string &legendposition) {
31:   // North-East
32:   x1 = 0.70;
33:   x2 = x1 + 0.18;
34:   y1 = 0.75 - 0.02 * N;
35:
36:   // South-East
37:   if (legendposition.compare("southeast") == 0) { y1 = 0.10 - 0.01 * N; }
38:   y2 = y1 + 0.04 * N;  // Scale by the number of histograms
39: }
40:
41: hlMultiplet::hlMultiplet(const std::string &name, const std::string &labeltext, int N,
42:                          double minval, double maxval, const std::vector<std::string> &legendtext) {
43:   N_              = N;
44:   name_           = name;
45:   minval_         = minval;
46:   maxval_         = maxval;
47:   legendtext_     = legendtext;
48:   legendposition_ = "northeast";
49:
50:   // Initialize histogram vector container size
51:   h.resize(legendtext.size());
52:
53:   for (const auto &i : indices(h)) {
54:     h[i] = new TH1D(Form("%s_%lu", name.c_str(), i), labeltext.c_str(), N, minval, maxval);
55:     h[i]->Sumw2();  // Error saving on
56:   }
57: }
58:
59: // Histogram normalization
60: void hlMultiplet::NormalizeAll(const std::vector<double> &cross_section,
61:                                const std::vector<double> &multiplier) {
62:   for (const auto &i : indices(h)) {
63:     double scale = 1.0;
64:
65:     // Takes into account weighted and unweight (weight=1) filling
66:     // We take into account also under/overflow here with macro: (0, Nbins+1)
67:     const double integral = h[i]->Integral(0, h[i]->GetNbinsX() + 1);
68:     if (integral > 0) { scale /= integral; }
69:
70:     // Binwidth
71:     const int    bin      = 1;
72:     const double binwidth = h[i]->GetXaxis()->GetBinWidth(bin);
73:     if (binwidth > 0) { scale /= binwidth; }
74:
75:     // Cross Section
76:     if (cross_section[i] > 0) { scale *= cross_section[i]; }
77:
78:     // Additional scale factor
79:     scale *= multiplier[i];
80:
81:     h[i]->Scale(scale);
82:   }
83: }
```

```
84:
85: std::vector<double> h1Multiplet::SaveFig(const std::string &fullpath, bool RATIOPLOT) const {
86:   // New colorscheme
87:   std::vector<int>                    color;
88:   std::vector<std::shared_ptr<TColor>> rootcolor;
89:   rootstyle::CreateColorMap(color, rootcolor);
90:
91:   // ------------------------------------------------------
92:   // Apply the chi2 test and retrieve the residuals
93:   gra::aux::PrintBar("*");
94:   std::vector<double> chi2ndf(h.size(), 0.0);
95:
96:   for (const auto &i : indices(h)) {
97:     double res[N_];
98:     printf("%s [%lu] :: \n", legendtext_[i].c_str(), i);
99:     double c2ndf = h[0]->Chi2Test(h[i], "WW P CHI2/NDF", res);
100:    chi2ndf[i]   = c2ndf;
101:
102:    printf("chi2/ndf = %0.3f \n\n", c2ndf);
103:  }
104:  gra::aux::PrintBar("*");
105:
106:  TCanvas c0("c", "c", 750, 800);
107:
108:  // Upper plot will be in pad1
109:  std::shared_ptr<TPad> pad1;
110:
111:  if (RATIOPLOT) {
112:    pad1 = std::make_shared<TPad>("pad1", "pad1", 0, 0.3, 1.0, 1.0);
113:    pad1->SetBottomMargin(0.015);  // Upper and lower plot are joined
114:  } else {
115:    pad1 = std::make_shared<TPad>("pad1", "pad1", 0, 0.0, 1.0, 1.0);
116:  }
117:  // pad1->SetGridx();          // Vertical grid
118:  pad1->Draw();       // Draw the upper pad: pad1
119:  pad1->cd();         // pad1 becomes the current pad
120:  h[0]->SetStats(0);  // No statistics on upper plot
121:
122:
123:  // Find maximum value for y-range limits
124:  double MAXVAL = 0.0;
125:  for (const auto &i : indices(h)) {
126:    const double max = h[i]->GetMaximum();
127:    if (max > MAXVAL) { MAXVAL = max; }
128:  }
129:
130:  // Find minimum (non-zero) value for y-range limits
131:  double MINVAL = 1e32;
132:  for (const auto &i : indices(h)) {
133:    for (int k = 0; k < h[i]->GetNbinsX(); ++k) {
134:      double value = h[i]->GetBinContent(k);
135:      if (value < MINVAL && value > 0) { MINVAL = value; }
136:    }
137:  }
138:
139:  // Loop over histograms
140:  for (const auto &i : indices(h)) {
141:    h[i]->SetLineColor(color[i]);
142:    h[i]->SetLineWidth(2);
143:    h[i]->SetMarkerColor(color[i]);
144:    h[i]->SetMarkerStyle(20 + i);
145:    h[i]->SetMarkerSize(0.73);
146:
147:    std::cout << MINVAL << std::endl;
148:    h[i]->GetYaxis()->SetRangeUser(0, MAXVAL * 1.35);
149:
150:    // h[i]->Draw("e2 same");   // Needs this combo to draw box-lines
151:    h[i]->Draw("hist same");
152:
153:    // h[i]->SetFillStyle(4050);
154:    // h[i]->SetFillColor(color[i]);
155:    // h[i]->SetFillColorAlpha(color[i], 0.1);
156:    // h[i]->Draw("f hist same"); // Filled
157:  }
158:
159:  pad1->RedrawAxis();  // Fix overlapping histogram lines on borders
160:
161:  // ------------------------------------------------------------------
162:  // Create dynamically sized legend (depending on number of legend entries)
163:
164:  double x1, x2, y1, y2 = 0.0;
165:  GetLegendPosition(h.size(), x1, x2, y1, y2, legendposition_);
166:  std::shared_ptr<TLegend> legend = std::make_shared<TLegend>(x1, y1, x2, y2);
```



```
167:  legend->SetFillColor(0);  // White background
168:  // legend->SetBorderSize(0); // No box
169:
170:  // Add legend entries
171:  for (const auto &i : indices(h)) { legend->AddEntry(h[i], legendtext_[i].c_str()); }
172:  legend->Draw();
173:
174:  // ------------------------------------------------------------------
175:  // Create labels
176:  std::shared_ptr<TLatex> l1;
177:  std::shared_ptr<TLatex> l2;
178:  gra::rootstyle::MadeInFinland(l1, l2, 0.935, ((RATIOPLOT == true ? 0.03 : 0.16), 0.58));
179:
180:  // ------------------------------------------------------------------
181:  // Ratio plots
182:  std::vector<TH1D *>     hR(h.size(), nullptr);
183:  std::shared_ptr<TPad>  pad2;
184:  std::shared_ptr<TLine> line;
185:
186:  if (RATIOPLOT) {
187:    c0.cd();
188:    pad2 = std::make_shared<TPad>("pad2", "pad2", 0, 0.05, 1, 0.3);
189:    pad2->SetTopMargin(0.025);
190:    pad2->SetBottomMargin(0.25);
191:    pad2->SetGridx();  // vertical grid
192:    pad2->Draw();
193:    pad2->cd();  // pad2 becomes the current pad
194:
195:    for (const auto &i : indices(h)) {
196:      hR[i] = static_cast<TH1D *>(h[i]->Clone(Form("ratio_%lu", i)));
197:      hR[i]->Divide(h[0]);
198:
199:      hR[i]->SetMinimum(0.0);  // y-axis range
200:      hR[i]->SetMaximum(2.0);  //
201:      hR[i]->SetStats(0);      // statistics box off
202:      hR[i]->Draw("same");     // ratio plot
203:
204:      // Ratio plot (h3) settings
205:      hR[i]->SetTitle("");  // Remove the ratio title
206:
207:      // Y axis ratio plot settings
208:      hR[i]->GetYaxis()->SetTitle("Ratio");
209:      hR[i]->GetYaxis()->CenterTitle();
210:      hR[i]->GetYaxis()->SetNdivisions(505);
211:      hR[i]->GetYaxis()->SetTitleSize(22);
212:      hR[i]->GetYaxis()->SetTitleFont(43);
213:      hR[i]->GetYaxis()->SetTitleOffset(1.55);
214:      hR[i]->GetYaxis()->SetLabelFont(45);  // Absolute font size in pixel (precision 3)
215:      hR[i]->GetYaxis()->SetLabelSize(15);
216:
217:      // X axis ratio plot settings
218:      hR[i]->GetXaxis()->SetTitleSize(22);
219:      hR[i]->GetXaxis()->SetTitleFont(43);
220:      hR[i]->GetXaxis()->SetTitleOffset(4.);
221:      hR[i]->GetXaxis()->SetLabelFont(45);  // Absolute font size in pixel (precision 3)
222:      hR[i]->GetXaxis()->SetLabelSize(18);
223:    }
224:
225:    // Draw horizontal line
226:    const double ymax = 1.0;
227:    line              = std::make_shared<TLine>(minval_, ymax, maxval_, ymax);
228:    line->SetLineColor(15);
229:    line->SetLineWidth(2.0);
230:    line->Draw();
231:
232:    // ------------------------------------------------------------------
233:    // Remove x-axis of UPPER PLOT
234:    h[0]->GetXaxis()->SetLabelOffset(999);
235:    h[0]->GetXaxis()->SetLabelSize(0);
236:    // ------------------------------------------------------------------
237:
238:  }  // RATIOPLOT
239:
240:  // Give y-axis title some offset to avoid overlapping with numbers
241:  h[0]->GetYaxis()->SetTitleOffset(1.45);
242:
243:  // Create output directory if it does not exist
244:  aux::CreateDirectory(fullpath);
245:
246:  // Save pdf
247:  std::string fullfile = fullpath + name_ + ".pdf";
248:  c0.SaveAs(fullfile.c_str());
249:
```



```
250:  // Save logscale pdf
251:  if (MINVAL > 0) {
252:    pad1->cd()->SetLogy();  // pad2 becomes the current pad
253:    for (const auto &i : indices(h)) {
254:      h[i]->GetYaxis()->SetRangeUser(std::max(MINVAL, MAXVAL * 1e-8), MAXVAL * 5);
255:    }
256:    fullfile = fullpath + name_ + "_logy" + ".pdf";
257:    c0.SaveAs(fullfile.c_str());
258:  }
259:
260:  // Remove histograms
261:  if (RATIOPLOT) {
262:    for (const auto &i : indices(hR)) { delete hR[i]; }
263:  }
264:
265:  return chi2ndf;
266: }
267:
268: h2Multiplet::h2Multiplet(const std::string &name, const std::string &labeltext, int N1,
269:                          double minval1, double maxval1, int N2, double minval2, double maxval2,
270:                          const std::vector<std::string> &legendtext) {
271:   name_ = name;
272:
273:   N1_ = N1;
274:   N2_ = N2;
275:
276:   minval1_ = minval1;
277:   maxval1_ = maxval1;
278:   minval2_ = minval2;
279:   maxval2_ = maxval2;
280:
281:   legendtext_ = legendtext;
282:
283:   // Initialize histogram vector container size
284:   h.resize(legendtext.size());
285:
286:   for (const auto &i : indices(h)) {
287:     h[i] = new TH2D(Form("%s_%lu", name.c_str(), i), labeltext.c_str(), N1, minval1, maxval1, N2,
288:                     minval2, maxval2);
289:     h[i]->Sumw2();  // Error saving on
290:   }
291: }
292:
293: void h2Multiplet::NormalizeAll(const std::vector<double> &cross_section,
294:                                const std::vector<double> &multiplier) {
295:   for (const auto &i : indices(h)) {
296:     double scale = 1.0;
297:
298:     // Takes into account weighted and unweight (weight=1) filling
299:     // We take into account also under/overflow here with macro: (0, Nbins+1)
300:     const double integral = h[i]->Integral(0, h[i]->GetNbinsX() + 1, 0, h[i]->GetNbinsY() + 1);
301:     if (integral > 0) { scale /= integral; }
302:
303:     // Binwidth
304:     const int    bin      = 1;
305:     const double binwidth = h[i]->GetXaxis()->GetBinWidth(bin) * h[i]->GetYaxis()->GetBinWidth(bin);
306:     if (binwidth > 0) { scale /= binwidth; }
307:
308:     // Cross Section
309:     if (cross_section[i] > 0) { scale *= cross_section[i]; }
310:
311:     // Additional scale factor
312:     scale *= multiplier[i];
313:
314:     h[i]->Scale(scale);
315:   }
316: }
317:
318: double h2Multiplet::SaveFig(const std::string &fullpath, bool RATIOPLOT) const {
319:   const double scale = (RATIOPLOT == true) ? 2.0 : 1.0;
320:
321:   TCanvas c0("c", "c", h.size() == 1 ? 520 : 750 / 3.0 * h.size(),
322:              260 * scale);  // scale canvas according to number of sources
323:   c0.Divide(h.size(), h.size() == 1 ? 1 : scale, 0.01, 0.02);  // [columns] x [rows]
324:
325:   // Normalize by the first histogram
326:   // const double ZMAX = h[0]->GetMaximum();
327:
328:   // Histograms on TOP ROW
329:   for (const auto &i : indices(h)) {
330:     c0.cd(i + 1);  // choose position
331:     c0.cd(i + 1)->SetRightMargin(0.13);
332:
```



```
333:    h[i]->SetStats(0);
334:    h[i]->Draw("COLZ");
335:    h[i]->GetYaxis()->SetTitleOffset(1.3);
336:    // h[i]->GetZaxis()->SetRangeUser(0.0, ZMAX);
337:    h[i]->SetTitle(legendtext_[i].c_str());
338:  }
339:
340:  // Ratio histograms on BOTTOM ROW
341:  std::vector<TH2D *> hR(h.size(), nullptr);
342:
343:  if (RATIOPLOT) {
344:    if (h.size() > 1) {  // Only if we have more than 1
345:
346:      for (const auto &i : indices(h)) {
347:        c0.cd(i + 1 + h.size());  // Choose position
348:        c0.cd(i + 1 + h.size())->SetRightMargin(0.13);
349:
350:        hR[i] = static_cast<TH2D *>(h[i]->Clone(Form("h2R_%lu", i)));
351:
352:        hR[i]->Divide(h[0]);  // Divide by 0-th histogram
353:        hR[i]->GetYaxis()->SetTitleOffset(1.3);
354:        hR[i]->SetStats(0);  // No statistics on upper plot
355:        hR[i]->Draw("COLZ");
356:        hR[i]->GetZaxis()->SetRangeUser(0.0, 2.0);
357:        hR[i]->SetTitle(Form("Ratio: %s / %s", legendtext_[i].c_str(), legendtext_[0].c_str()));
358:      }
359:    }
360:  }
361:
362:  // ------------------------------------------------------------------
363:  // Create labels
364:  c0.cd();  // Important!
365:  std::shared_ptr<TPad> tpad;
366:  gra::rootstyle::TransparentPad(tpad);
367:
368:  std::shared_ptr<TLatex> l1;
369:  std::shared_ptr<TLatex> l2;
370:  gra::rootstyle::MadeInFinland(l1, l2, 0.99);
371:  // ------------------------------------------------------------------
372:
373:  // Create output directory if it does not exist
374:  aux::CreateDirectory(fullpath);
375:
376:  // Save pdf
377:  std::string fullfile = fullpath + name_ + ".pdf";
378:  c0.SaveAs(fullfile.c_str());
379:
380:  if (RATIOPLOT) {
381:    // Delete ratio histograms from memory
382:    if (h.size() > 1) {
383:      for (const auto &i : indices(hR)) { delete hR[i]; }
384:    }
385:  }
386:
387:  return 0.0;
388: }
389:
390:
391: hProfMultiplet::hProfMultiplet(const std::string &name, const std::string &labeltext, int N,
392:                                double minval1, double maxval1, double minval2, double maxval2,
393:                                const std::vector<std::string> &legendtext) {
394:   name_ = name;
395:
396:   N_       = N;
397:   minval1_ = minval1;
398:   maxval1_ = maxval1;
399:
400:   minval2_ = minval2;
401:   maxval2_ = maxval2;
402:
403:   legendtext_     = legendtext;
404:   legendposition_ = "northeast";
405:
406:   // Initialize histogram vector container size
407:   h.resize(legendtext.size());
408:
409:   for (const auto &i : indices(h)) {
410:     h[i] = new TProfile(Form("%s_%lu", name.c_str(), i), labeltext.c_str(), N, minval1, maxval1,
411:                         minval2, maxval2);
412:     h[i]->Sumw2();  // Error saving on
413:   }
414: }
415:
```

./src/Analysis/MMultiplet.cc 6/8

```
416: double hProfMultiplet::SaveFig(const std::string &fullpath, bool RATIOPLOT) const {
417:   // New colorscheme
418:   std::vector<int>                color;
419:   std::vector<std::shared_ptr<TColor>> rootcolor;
420:   rootstyle::CreateColorMap(color, rootcolor);
421:
422:   TCanvas c0("c", "c", 750, 800);
423:
424:   // Upper plot will be in pad1
425:   std::shared_ptr<TPad> pad1;
426:
427:   if (RATIOPLOT) {
428:     pad1 = std::make_shared<TPad>("pad1", "pad1", 0, 0.3, 1.0, 1.0);
429:     pad1->SetBottomMargin(0.015);  // Upper and lower plot are joined
430:   } else {
431:     pad1 = std::make_shared<TPad>("pad1", "pad1", 0, 0.0, 1.0, 1.0);
432:   }
433:   // pad1->SetGridx();           // Vertical grid
434:   pad1->Draw();       // Draw the upper pad: pad1
435:   pad1->cd();         // pad1 becomes the current pad
436:   h[0]->SetStats(0);  // No statistics on upper plot
437:
438:   // Find maximum value for y-range limits
439:   double MAXVAL = 0.0;
440:   double MINVAL = 1e32;
441:   for (const auto &i : indices(h)) {
442:     if (h[i]->GetMaximum() > MAXVAL) { MAXVAL = h[i]->GetMaximum(); }
443:     if (h[i]->GetMinimum() < MINVAL) { MINVAL = h[i]->GetMinimum(); }
444:   }
445:
446:   // Loop over histograms
447:   for (const auto &i : indices(h)) {
448:     h[i]->SetLineColor(color[i]);
449:     h[i]->SetMarkerColor(color[i]);
450:     h[i]->SetMarkerStyle(20);
451:     h[i]->SetMarkerSize(0.5);
452:     h[i]->GetYaxis()->SetRangeUser(MINVAL, MAXVAL * 1.25);
453:
454:     h[i]->Draw("L SAME");
455:   }
456:
457:   // -----------------------------------------------------------------
458:   // LEGEND
459:   // North-East
460:   double x1, x2, y1, y2 = 0.0;
461:   GetLegendPosition(h.size(), x1, x2, y1, y2, legendposition_);
462:   std::shared_ptr<TLegend> legend = std::make_shared<TLegend>(x1, y1, x2, y2);
463:
464:   legend->SetFillColor(0);  // White background
465:   // legend->SetBorderSize(0); // No box
466:
467:   // Add legend entries
468:   for (const auto &i : indices(h)) { legend->AddEntry(h[i], legendtext_[i].c_str()); }
469:   legend->Draw()
470:
471:   // -----------------------------------------------------------------
472:   // GRANIITTI text
473:   std::shared_ptr<TLatex> l1;
474:   std::shared_ptr<TLatex> l2;
475:   gra::rootstyle::MadeInFinland(l1, l2, 0.935, ((RATIOPLOT == true ? 0.03 : 0.16), 0.58));
476:   // -----------------------------------------------------------------
477:
478:   // -----------------------------------------------------------------
479:   // Ratio plots
480:   std::vector<TProfile *> hxP(h.size(), nullptr);
481:   std::vector<TH1D *>     hR(h.size(), nullptr);
482:   TH1D *                  hx0 = nullptr;
483:   std::shared_ptr<TPad>   pad2;
484:   std::shared_ptr<TLine>  line;
485:
486:   if (RATIOPLOT) {
487:     c0.cd();
488:     pad2 = std::make_shared<TPad>("pad2", "pad2", 0, 0.05, 1, 0.3);
489:     pad2->SetTopMargin(0.025);
490:     pad2->SetBottomMargin(0.25);
491:     pad2->SetGridx();  // vertical grid
492:     pad2->Draw();
493:     pad2->cd();  // pad2 becomes the current pad
494:
495:     for (const auto &i : indices(h)) {
496:       hxP[i] = static_cast<TProfile *>(h[i]->Clone(Form("ratio_%lu", i)));
497:
498:       // -------------------------------------------------------
```

./src/Analysis/MMultiplet.cc 7/8

```
499:       // To get proper errors, we need to use TH1 instead of TProfile
500:       hR[i] = hxP[i]->ProjectionX();
501:
502:       if (i == 0) {  // this is needed as the denominator
503:         hx0 = static_cast<TH1D *>(hxP[i]->Clone("denominator"));
504:       }
505:
506:       // Set same colors as above
507:       hR[i]->SetLineColor(color[i]);
508:       hR[i]->SetMarkerColor(color[i]);
509:       hR[i]->SetMarkerStyle(20);
510:       hR[i]->SetMarkerSize(0.5);
511:
512:       // -------------------------------------------------------------
513:
514:       // Get ratio to 0-th histogram
515:       hR[i]->Divide(hx0);
516:
517:       hR[i]->SetMinimum(0.0);  // y-range
518:       hR[i]->SetMaximum(2.0);  //
519:       hR[i]->SetStats(0);      // no statistics box
520:       hR[i]->Draw("same");     // ratio plot
521:
522:       // Ratio plot (h3) settings
523:       hR[i]->SetTitle("");  // remove title
524:
525:       // Y axis ratio plot settings
526:       hR[i]->GetYaxis()->SetTitle("Ratio");
527:       hR[i]->GetYaxis()->CenterTitle();
528:       hR[i]->GetYaxis()->SetNdivisions(505);
529:       hR[i]->GetYaxis()->SetTitleSize(20);
530:       hR[i]->GetYaxis()->SetTitleFont(43);
531:       hR[i]->GetYaxis()->SetTitleOffset(1.55);
532:       hR[i]->GetYaxis()->SetLabelFont(43);  // in pixel (precision 3)
533:       hR[i]->GetYaxis()->SetLabelSize(15);
534:
535:       // X axis ratio plot settings
536:       hR[i]->GetXaxis()->SetTitleSize(20);
537:       hR[i]->GetXaxis()->SetTitleFont(43);
538:       hR[i]->GetXaxis()->SetTitleOffset(4.);
539:       hR[i]->GetXaxis()->SetLabelFont(43);  // in pixel (precision 3)
540:       hR[i]->GetXaxis()->SetLabelSize(15);
541:     }
542:
543:     // Draw horizontal line
544:     const double ymax = 1.0;
545:     line              = std::make_shared<TLine>(minval1_, ymax, maxval1_, ymax);
546:     line->SetLineColor(15);
547:     line->SetLineWidth(2.0);
548:     line->Draw();
549:
550:     // -------------------------------------------------------------------
551:     // Remove x-axis of UPPER PLOT
552:     h[0]->GetXaxis()->SetLabelOffset(999);
553:     h[0]->GetXaxis()->SetLabelSize(0);
554:     // -------------------------------------------------------------------
555:
556:   }  // RATIOPLOT
557:
558:   // Give y-axis title some offset to avoid overlapping with numbers
559:   h[0]->GetYaxis()->SetTitleOffset(1.45);
560:
561:   // Create output directory if it does not exist
562:   aux::CreateDirectory(fullpath);
563:
564:   // Save pdf
565:   std::string fullfile = fullpath + name_ + ".pdf";
566:   c0.SaveAs(fullfile.c_str());
567:
568:   // Save logscale pdf
569:   if (MINVAL > 0) {
570:     pad1->cd()->SetLogy();  // pad2 becomes the current pad
571:     fullfile = fullpath + name_ + "_logy" + ".pdf";
572:     c0.SaveAs(fullfile.c_str());
573:   }
574:
575:   if (RATIOPLOT) {
576:     // Remove histograms
577:     for (const auto &i : indices(hR)) {
578:       delete hxP[i];
579:       delete hR[i];
580:     }
581:     delete hx0;
```

./src/Analysis/MMultiplet.cc 8/8

```
582:   }
583:
584:   return 0.0;
585: }
586:
587: }  // namespace gra
588:
```

./src/MEikonal.cc 1/7

```
  1: // Eikonal density and screening class
  2: //
  3: //
  4: // (c) 2017-2020 Mikael Mieskolainen
  5: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  6:
  7: // C++
  8: #include <cmath>
  9: #include <fstream>
 10: #include <future>
 11: #include <iomanip>
 12: #include <iostream>
 13: #include <vector>
 14:
 15: // Own
 16: #include "Graniitti/MAux.h"
 17: #include "Graniitti/MEikonal.h"
 18: #include "Graniitti/MForm.h"
 19: #include "Graniitti/MMath.h"
 20: #include "Graniitti/MPDG.h"
 21: #include "Graniitti/MTimer.h"
 22:
 23: // Libraries
 24: #include "extmath.hpp"
 25: #include "json.hpp"
 26: #include "rang.hpp"
 27:
 28: using gra::aux::indices;
 29: using gra::math::msqrt;
 30: using gra::math::pow2;
 31: using gra::math::zi;
 32:
 33: namespace gra {
 34:
 35: MEikonal::MEikonal() {}
 36: MEikonal::~MEikonal() {}
 37:
 38: // Return total, elastic, inelastic cross sections
 39: void MEikonal::GetTotXS(double &tot, double &el, double &in) const {
 40:   tot = sigma_tot;
 41:   el  = sigma_el;
 42:   in  = sigma_inel;
 43: }
 44:
 45: // Construct density and amplitude
 46: void MEikonal::S3Constructor(double s_in, const std::vector<gra::MParticle> &initialstate_in,
 47:                              bool onlydensity, int NumberBT, int NumberKT2) {
 48:   std::cout << "MEikonal::S3Constructor:" << std::endl;
 49:
 50:   // This first
 51:   Numerics.ReadParameters();
 52:
 53:   // Override
 54:   if (NumberBT > 0) { Numerics.NumberBT = NumberBT; }
 55:   if (NumberKT2 > 0) { Numerics.NumberKT2 = NumberKT2; }
 56:
 57:   // Mandelstam s and initial state
 58:   s                = s_in;
 59:   INITIALSTATE = initialstate_in;
 60:
 61:   // -------------------------------------------------------------------------
 62:   // Calculate hash based on all free variables -> if something changed,
 63:   // calculate new densities
 64:
 65:   // Proton density
 66:   {
 67:     std::cout << "Initializing <eikonal density> array:" << std::endl;
 68:     MBT.sqrts = msqrt(s);  // FIRST THIS
 69:     MBT.Set("bt", Numerics.MinBT, Numerics.MaxBT, Numerics.NumberBT, Numerics.logBT);
 70:     MBT.InitArray();  // Initialize (call last!)
 71:
 72:     const std::string hstr = std::to_string(s) + std::to_string(INITIALSTATE[0].pdg) + "_" +
 73:                              std::to_string(INITIALSTATE[1].pdg) + PARAM_SOFT::GetHashString() +
 74:                              Numerics.GetHashString();
 75:     const unsigned long hash     = gra::aux::djb2hash(hstr);
 76:     const std::string   filename = gra::aux::GetBasePath(2) + "/eikonal/" + "MBT_" +
 77:                                    std::to_string(INITIALSTATE[0].pdg) + "_" +
 78:                                    std::to_string(INITIALSTATE[1].pdg) + "_" +
 79:                                    gra::aux::ToString(msqrt(s), 0) + "_" + std::to_string(hash);
 80:
 81:     // Try to read pre-calculated
 82:     bool ok = MBT.ReadArray(filename);
 83:
```

```cpp
    while (!ok) {  // Problem, re-calculate
      // Pointer to member function: ReturnType
      // (ClassType::*)(ParameterTypes...)
      std::complex<double> (MEikonal::*f)(double) const = &MEikonal::S3Density;
      S3CalculateArray(MBT, f);
      MBT.WriteArray(filename, true);
      ok = MBT.ReadArray(filename);
    }
  }

  // Calculate cross sections (is fast)
  S3CalcXS();

  // Init cut Pomerons (is fast)
  S3InitCutPomerons();

  // Amplitude
  if (onlydensity == false) {
    std::cout << "Initializing <eikonal amplitude> array:" << std::endl;

    MSA.sqrts = msqrt(s);  // FIRST THIS
    MSA.Set("kt2", Numerics.MinKT2, Numerics.MaxKT2, Numerics.NumberKT2, Numerics.logKT2);
    MSA.InitArray();  // Initialize (call last!)

    const std::string hstr = std::to_string(s) + std::to_string(INITIALSTATE[0].pdg) +
                             std::to_string(INITIALSTATE[1].pdg) + PARAM_SOFT::GetHashString() +
                             Numerics.GetHashString();

    const unsigned long hash     = gra::aux::djb2hash(hstr);
    const std::string   filename = gra::aux::GetBasePath(2) + "/eikonal/" + "MSA_" +
                                   std::to_string(INITIALSTATE[0].pdg) + "_" +
                                   std::to_string(INITIALSTATE[1].pdg) + "_" +
                                   gra::aux::ToString(msqrt(s), 0) + "_" + std::to_string(hash);

    // Try to read pre-calculated
    bool ok = MSA.ReadArray(filename);

    while (!ok) {  // Problem, re-calculate
      // Pointer to member function: ReturnType
      // (ClassType::*)(ParameterTypes...)
      std::complex<double> (MEikonal::*f)(double) const = &MEikonal::S3Screening;
      S3CalculateArray(MSA, f);
      MSA.WriteArray(filename, true);
      ok = MSA.ReadArray(filename);
    }
  }
  // Tag it done
  S3INIT = true;
}

// Single elastic-pp Pomeron (Odderon) exchange amplitude
//
// This is used within eikonalization procedure
//
//
std::complex<double> MEikonal::SingleAmpElastic(double s, double t, int type) {
  // Proton form factors (could be here extended to multichannel)
  const double F_i = gra::form::S3F(t);
  const double F_k = gra::form::S3F(t);

  // Pomeron trajectory alpha(t), the same for Odderon
  const double alpha_P = gra::form::S3PomAlpha(t);

  // Pomeron exchange amplitude:
  // [Regge signature x Proton Form Factor x coupling x Proton
  //  Form Factor x coupling x Propagator ]
  const double s0 = 1.0;  // Typical (normalization) scale GeV^{-2}

  if (type == 1) {  // Pomeron (C-even)
    return gra::math::pow2(PARAM_SOFT::gN_P) * F_i * F_k * gra::form::ReggeEta(alpha_P, 1) *
           std::pow(s / s0, alpha_P);
  } else if (type == -1) {  // Odderon (C-even)
    return gra::math::pow2(PARAM_SOFT::gN_O) * F_i * F_k * gra::form::ReggeEta(alpha_P, -1) *
           std::pow(s / s0, alpha_P);
  } else {
    throw std::invalid_argument("MEikonal::SingleAmpElastic: Unknown input type " +
                                std::to_string(type));
  }
}


// Proton bt-density Omega(s,b_t) by Fourier-Bessel transform of the t-density
// see <mathworld.wolfram.com/HankelTransform.html>
```



```cpp
//
//
// For eikonalization see:
// [REFERENCE: Desgrolard, Giffon, Martynov, Predazzi, arxiv.org/abs/hep-ph/9907451v2]
// [REFERENCE: Desgrolard, Giffon, Martynov, Predazzi, arxiv.org/abs/hep-ph/0001149]
//
//
// For some discussion about Odderon versus Pomeron, see:
// [REFERENCE: Ewerz, Maniatis, Nachtmann, arxiv.org/abs/1309.3478]
//
std::complex<double> MEikonal::S3Density(double bt) const {
  // Discretization of kt
  const double kt_STEP =
      (Numerics.FBIntegralMaxKT - Numerics.FBIntegralMinKT) / Numerics.FBIntegralN;

  // N + 1!
  std::vector<std::complex<double>> f(Numerics.FBIntegralN + 1, 0.0);

  // Initial state configuration [NOT IMPLEMENTED = SAME FOR pp and ppbar]
  // pp
  if (INITIALSTATE[0].pdg == PDG::PDG_p && INITIALSTATE[1].pdg == PDG::PDG_p) {
    // TBD
    // ppbar
  } else if ((INITIALSTATE[0].pdg == PDG::PDG_p && INITIALSTATE[1].pdg == -PDG::PDG_p) ||
             (INITIALSTATE[0].pdg == -PDG::PDG_p && INITIALSTATE[1].pdg == PDG::PDG_p)) {
    // TBD
  }

  // Loop over
  for (const auto &i : indices(f)) {
    const double kt = Numerics.FBIntegralMinKT + i * kt_STEP;

    // Negative, with Mandelstam t ~= -kt^2
    const double t = -gra::math::pow2(kt);

    // Pomeron exchange (positive signature)
    std::complex<double> A = SingleAmpElastic(s, t, 1);

    // Odderon exchange (negative signature)
    if (PARAM_SOFT::ODDERON_ON == true) { A += SingleAmpElastic(s, t, -1); }

    // Value
    f[i] = A * BESSJ0(bt * kt) * kt;
    // f[i] = A * std::cyl_bessel_j(0, bt * kt) * kt; // c++17, slow
  }
  // [2pi from Bessel phi-integral] / [ (2pi)^2] (2D-Fourier factor) * s ]
  const double FACTOR = 1 / (2.0 * gra::math::PI * s);

  return gra::math::CSIntegral(f, kt_STEP) * FACTOR;
}

// Calculate eikonalized elastic screening amplitude: A_eik(s,t=-kt^2)
// in kt^2 space via bt-space Fourier-Bessel integral
//
//
std::complex<double> MEikonal::S3Screening(double kt2) const {
  // Local discretization
  const double STEP = (Numerics.MaxBT - Numerics.MinBT) / Numerics.FBIntegralN;

  const double                      kt = gra::math::msqrt(kt2);
  std::vector<std::complex<double>> f(Numerics.FBIntegralN + 1, 0.0);

  // Numerical integral loop over impact parameter (b_t) space
  for (const auto &i : indices(f)) {
    const double               bt    = Numerics.MinBT + i * STEP;
    const std::complex<double> Omega = MBT.Interpolate1D(bt);

    // I. STANDARD EIKONAL APPROXIMATION
    const std::complex<double> A = gra::math::zi * (1.0 - std::exp(gra::math::zi * Omega / 2.0));

    f[i] = A * BESSJ0(bt * kt) * bt;
    // f[i] = A * std::cyl_bessel_j(0, bt * kt) * bt; // c++17, slow
  }
  // Bessel phi-integral factor
  const double FACTOR = 2.0 * gra::math::PI;

  // Numerical integration
  return (2.0 * s) * FACTOR * gra::math::CSIntegral(f, STEP);
}

// Calculate screened total, elastic and inelastic cross sections
// in the eikonal model
//
```



```cpp
//
// \int d^2b f(b) = \int_0^\inf \int_0^{2\pi} r dr d\theta f(r,\theta)
//                = 2\pi \int_0^\inf f(r) r dr
//
void MEikonal::S3CalcXS() {
  std::cout << "MEikonal::S3CalcXS:" << std::endl << std::endl;

  // Local discretization
  const unsigned int N    = 2 * 3000;  // even number
  const double       STEP = (Numerics.MaxBT - Numerics.MinBT) / N;

  // Two channel eikonal eigenvalue solutions obtained via symbolic
  // calculation see e.g.
  //
  // [REFERENCE: Khoze, Martin, Ryskin, arxiv.org/abs/hep-ph/0007359v2]
  // [REFERENCE: Roehr, inspirehep.net/record/1351489/files/Thesis-2014-Roehr.pdf]
  //
  // Unitary transformation matrix
  const MMatrix<double> cc = {{1, 1, 1, 1},     // pp
                              {-1, 1, 1, -1},   // pN*
                              {-1, 1, -1, 1},   // N*p
                              {1, 1, -1, -1}};  // N*N*
  // Eigenvalues
  const std::vector<double> lambda = {
      gra::math::pow2(1.0 - PARAM_SOFT::gamma), gra::math::pow2(1.0 + PARAM_SOFT::gamma),
      1.0 - gra::math::pow2(PARAM_SOFT::gamma), 1.0 - gra::math::pow2(PARAM_SOFT::gamma)};
  // ------------------------------------------------------------------------------

  // N+1
  std::vector<std::complex<double>> f_tot(N + 1, 0.0);
  std::vector<std::complex<double>> f_el(N + 1, 0.0);
  std::vector<std::complex<double>> f_in(N + 1, 0.0);

  // Two-channel eikonal, N+1!
  std::vector<std::vector<std::complex<double>>> f_2(4,
                                                     std::vector<std::complex<double>>(N + 1, 0.0));

  // Numerical integral loop over impact parameter (b_t) space
  for (const auto &i : indices(f_tot)) {
    const double bt = Numerics.MinBT + i * STEP;

    // Calculate density
    const std::complex<double> Omega = MBT.Interpolate1D(bt);

    // ------------------------------------------------------------------------
    // Single-Channel eikonal

    // Elastic amplitude A_el(s,b)
    const std::complex<double> A_el = gra::math::zi * (1.0 - std::exp(gra::math::zi * Omega / 2.0));

    // TOTAL: 2.0 * Im A_el(s,b)
    f_tot[i] = 2.0 * std::imag(A_el);

    // ELASTIC: |A_el(s,b)|^2
    f_el[i] = gra::math::abs2(A_el);

    // INELASTIC: 2Im A_el(s,b) - |A_el(s,b)|^2
    f_in[i] = 2.0 * std::imag(A_el) - gra::math::abs2(A_el);
    // ------------------------------------------------------------------------

    // ------------------------------------------------------------------------
    // Two-channel Eikonal solutions for pp->p(*)p(*)
    const std::vector<std::complex<double>> sol = {
        1.0 - std::exp(gra::math::zi * lambda[0] * Omega / 2.0),
        1.0 - std::exp(gra::math::zi * lambda[1] * Omega / 2.0),
        1.0 - std::exp(gra::math::zi * lambda[2] * Omega / 2.0),
        1.0 - std::exp(gra::math::zi * lambda[3] * Omega / 2.0)};

    for (std::size_t k = 0; k < 4; ++k) {
      // Amplitude squared
      f_2[k][i] = gra::math::abs2(
          (sol[0] * cc[k][0] + sol[1] * cc[k][1] + sol[2] * cc[k][2] + sol[3] * cc[k][3]) / 4.0);
      f_2[k][i] *= bt;  // Jacobian
    }

    // Jacobian
    f_tot[i] *= bt;
    f_el[i] *= bt;
    f_in[i] *= bt;
  }

  // 2D-Integral factor
  const double IC = 2.0 * gra::math::PI;
```



```cpp

  // Composite Simpson's rule, real is taken for C++ reasons in order to
  // be able to substitute into double
  sigma_tot  = IC * std::real(gra::math::CSIntegral(f_tot, STEP)) * PDG::GeV2barn;
  sigma_el   = IC * std::real(gra::math::CSIntegral(f_el, STEP)) * PDG::GeV2barn;
  sigma_inel = IC * std::real(gra::math::CSIntegral(f_in, STEP)) * PDG::GeV2barn;
  // sigma_inel = sigma_tot - sigma_el; // cross check

  // Comments for the "multichannel" formalism [next version]

  // Total cross section:    2*\int d^2b sum_ik |a_i|^2|a_k|^2 (1 -
  // exp^(-Omega(s,b)/2))
  std::cout << "Single Channel Eikonal:" << std::endl;
  printf(" Total xs:      %0.3f mb \n", sigma_tot * 1E3);

  // Elastic cross section:  \int d^2b (sum_ik |a_i|^2|a_k|^2 (1 -
  // exp^(-Omega(s,b)/2))^2
  printf(" Elastic xs:    %0.3f mb \n", sigma_el * 1E3);

  // Inelastic cross section:  \int d^2b sum_ik |a_i|^2|a_k|^2 (1 -
  // exp^(-Omega(s,b)/2))
  printf(" Inelastic xs:  %0.3f mb \n\n", sigma_inel * 1E3);

  const double sigma_el_2 = IC * std::real(gra::math::CSIntegral(f_2[0], STEP)) * PDG::GeV2barn;
  const double sigma_sd_a = IC * std::real(gra::math::CSIntegral(f_2[1], STEP)) * PDG::GeV2barn;
  const double sigma_sd_b = IC * std::real(gra::math::CSIntegral(f_2[2], STEP)) * PDG::GeV2barn;
  const double sigma_dd   = IC * std::real(gra::math::CSIntegral(f_2[3], STEP)) * PDG::GeV2barn;

  // Scale
  const double kappa = sigma_el / sigma_el_2;

  std::cout << "Two Channel normalized to Single Channel [PROTOTEST]" << std::endl;
  printf("Calculated k = <el1>/<el2> = %0.3f \n", kappa);

  sigma_diff[0] = sigma_el_2 * kappa * 1E3;
  sigma_diff[1] = sigma_sd_a * kappa * 1E3;
  sigma_diff[2] = sigma_sd_b * kappa * 1E3;
  sigma_diff[3] = sigma_dd * kappa * 1E3;

  printf(" pp   xs:  %0.3f mb \n", sigma_diff[0]);
  printf(" pN*  xs:  %0.3f mb \n", sigma_diff[1]);
  printf(" N*p  xs:  %0.3f mb \n", sigma_diff[2]);
  printf(" N*N* xs:  %0.3f mb \n\n", sigma_diff[3]);
}

// Calculate interpolation arrays
void MEikonal::S3CalculateArray(IArray1D &arr, std::complex<double> (MEikonal::*f)(double) const) {
  std::cout << "MEikonal::S3CalculateArray:" << std::endl;
  std::vector<std::future<std::complex<double>>> futures;  // std::async return values
  MTimer                                         timer(true);

  // Loop over discretized variable
  for (std::size_t i = 0; i < arr.F.size_row(); ++i) {
    const double a = arr.MIN + i * arr.STEP;
    arr.F[i][X]    = a;

    // Transform input to linear if log stepping, for the function
    const double var = (arr.islog) ? std::exp(a) : a;

    gra::aux::PrintProgress(i / static_cast<double>(arr.N + 1));
    futures.push_back(std::async(std::launch::async, f, this, var));
  }
  gra::aux::ClearProgress();
  std::cout << std::endl;
  printf("- Time elapsed: %0.1f sec \n\n", timer.ElapsedSec());

  // Retrieve std::async values
  for (const auto &i : indices(futures)) { arr.F[i][Y] = futures[i].get(); }
}

// Write the array to a file
bool IArray1D::WriteArray(const std::string &filename, bool overwrite) const {
  // Do not write if file exists already
  if (gra::aux::FileExist(filename) && !overwrite) {
    // std::cout << "- Found pre-calculated" << std::endl;
    return true;
  }
  std::ofstream file;
  file.open(filename);
  if (!file.is_open()) {
    std::string str = "IArray1D::WriteArray: Fatal IO-error with: " + filename;
    throw std::invalid_argument(str);
  }
```

./src/MEikonal.cc 6/7

```
416:
417:   MTimer timer(true);
418:   std::cout << "IArray1D::WriteArray: ";
419:   unsigned int line_number = 0;
420:
421:   try {
422:     for (std::size_t i = 0; i < F.size_row(); ++i) {
423:       // Write to file
424:       file << std::setprecision(15) << std::real(F[i][X]) << "," << std::real(F[i][Y]) << ","
425:            << std::imag(F[i][Y]) << std::endl;
426:       ++line_number;
427:     }
428:   } catch (...) {
429:     throw std::invalid_argument("IArray1D:WriteArray: Error in file " + filename + " at line " +
430:                                 std::to_string(line_number));
431:   }
432:
433:   printf("Time elapsed %0.1f sec \n", timer.ElapsedSec());
434:   file.close();
435:   return true;
436: }
437:
438: // Read the array from a file
439: bool IArray1D::ReadArray(const std::string &filename) {
440:   std::ifstream file;
441:   file.open(filename);
442:   if (!file.is_open()) {
443:     std::string str = "IArray1D::ReadArray: Fatal IO-error with: " + filename;
444:     return false;
445:   }
446:   std::string  line;
447:   unsigned int fills = 0;
448:   std::cout << "IArray1D::ReadArray: ";
449:   unsigned line_number = 0;
450:
451:   try {
452:     for (std::size_t i = 0; i < F.size_row(); ++i) {
453:       // Read every line from the stream
454:       getline(file, line);
455:
456:       std::istringstream  stream(line);
457:       std::vector<double> columns(3, 0.0);
458:       std::string         element;
459:
460:       // Get every line element (3 of them) separated by separator
461:       int k = 0;
462:       while (getline(stream, element, ',')) {
463:         columns[k] = std::stod(element);  // string to double
464:         ++k;
465:         ++fills;
466:       }
467:       F[i][X] = columns[0];
468:       F[i][Y] = std::complex<double>(columns[1], columns[2]);
469:
470:       ++line_number;
471:     }
472:   } catch (...) {
473:     throw std::invalid_argument("IArray1D:ReadArray: Error in file " + filename + " at line " +
474:                                 std::to_string(line_number));
475:   }
476:   file.close();
477:
478:   if (fills != 3 * (N + 1)) {
479:     std::string str = "Corrupted file: " + filename;
480:     std::cout << str << std::endl;
481:     return false;
482:   }
483:   std::cout << rang::fg::green << "[DONE]" << rang::fg::reset << std::endl;
484:   return true;
485: }
486:
487: // Standard 1D-linear interpolation
488: //
489: std::complex<double> IArray1D::InterpolateID(double a) const {
490:   const double EPS = 1e-5;
491:   if (a < MIN) { a = MIN; }  // Truncate before (possible) logarithm
492:
493:   // Logarithmic stepping or not
494:   if (islog) { a = std::log(a); }
495:
496:   if (a > MAX * (1 + EPS)) {
497:     printf(
498:         "IArray1D::InterpolateID(%s) Input out of grid domain: "
```

./src/MEikonal.cc 7/7

```
499:         "%s = %0.3f [%0.3f, %0.3f] \n",
500:         name.c_str(), name.c_str(), a, MIN, MAX);
501:
502:     throw std::invalid_argument("InterpolateID:: Out of grid domain");
503:   }
504:   int i = std::floor((a - MIN) / STEP);
505:
506:   // Boundary protection
507:   if (i < 0) { i = 0; }              // Int needed for this, instead of unsigned int
508:   if (i >= (int)N) { i = N - 1; }  // We got N+1 elements in F
509:
510:   // y = y0 + (x - x0)*[(y1 - y0)/(x1 - x0)]
511:   return F[i][Y] + (a - F[i][X]) * ((F[i + 1][Y] - F[i][Y]) / (F[i + 1][X] - F[i][X]));
512: }
513:
514: // Calculate the number of cut soft Pomerons for the inelastic
515: //
516: void MEikonal::S3InitCutPomerons() {
517:   std::cout << "MEikonal::S3InitCutPomerons: [PROTOTEST]" << std::endl;
518:
519:   // Numerical integral loop over impact parameter (b_t) space
520:   const double STEP = (Numerics.MaxBT - Numerics.MinBT) / Numerics.NumberBT;
521:   P_array = std::vector<std::vector<double>>(MCUT, std::vector<double>(Numerics.NumberBT + 1, 0.0));
522:
523:   for (std::size_t j = 0; j < Numerics.NumberBT + 1; ++j) {
524:     const double bt = Numerics.MinBT + j * STEP;
525:     const double XI = std::imag(MBT.InterpolateID(bt));
526:
527:     // Poisson probabilities P_m(bt)
528:     for (std::size_t m = 1; m < MCUT; ++m) {
529:       // Poisson ansatz
530:       double P_m    = std::pow(2 * XI, m) / gra::math::factorial(m) * std::exp(-2 * XI);
531:       P_array[m][j] = P_m * bt;  // *bt from jacobian \int d^2b ...
532:     }
533:   }
534:
535:   // Impact parameter <bt> average probabilities
536:   P_cut.resize(MCUT, 0.0);
537:   for (std::size_t m = 0; m < MCUT; ++m) {
538:     P_cut[m] = gra::math::cSIntegral(P_array[m], STEP) / (Numerics.MaxBT - Numerics.MinBT);
539:     printf("P_cut[m=%2lu] = %0.5f \n", m, P_cut[m]);
540:   }
541:   std::cout << "-----------------------------" << std::endl;
542:   printf("P_cut[SUM ] = %0.5f \n", std::accumulate(P_cut.begin(), P_cut.end(), 0.0));
543:
544:   // Calculate zero-truncated average
545:   double avg = 0;
546:   for (std::size_t m = 1; m < P_cut.size(); ++m) { avg += m * P_cut[m]; }
547:   printf("<P_cut[m>0]> = %0.2f \n\n", avg);
548: }
549:
550: }  // namespace gra
551:
```

./src/MParton.cc 1/4

```
  1: // Collinear parton model 2->2 or (2->N) type phase space class
  2: // for calculations without proton (remnant) considerations
  3: //
  4: // (c) 2017-2020 Mikael Mieskolainen
  5: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  6:
  7: // C++
  8: #include <algorithm>
  9: #include <complex>
 10: #include <iostream>
 11: #include <random>
 12: #include <vector>
 13:
 14: // Own
 15: #include "Granitti/MAux.h"
 16: #include "Granitti/MForm.h"
 17: #include "Granitti/MFragment.h"
 18: #include "Granitti/MKinematics.h"
 19: #include "Granitti/MMath.h"
 20: #include "Granitti/MMatrix.h"
 21: #include "Granitti/MParton.h"
 22: #include "Granitti/MProcess.h"
 23: #include "Granitti/MSpin.h"
 24: #include "Granitti/MUserCuts.h"
 25:
 26: // Libraries
 27: #include "rang.hpp"
 28:
 29: using gra::aux::indices;
 30:
 31: using gra::math::CheckEMC;
 32: using gra::math::rPI;
 33: using gra::math::abs2;
 34: using gra::math::msqrt;
 35: using gra::math::pow2;
 36: using gra::math::pow3;
 37: using gra::math::pow4;
 38: using gra::math::pow5;
 39: using gra::math::zi;
 40:
 41: using gra::PDG::GeV2barn;
 42:
 43: namespace gra {
 44: // This is needed by construction
 45: MParton::MParton() { Initialize(); }
 46:
 47: // Constructor
 48: MParton::MParton(std::string process, const std::vector<aux::OneCMD> &syntax) {
 49:   Initialize();
 50:   InitHistograms();
 51:   SetProcess(process, syntax);
 52:
 53:   // Init final states
 54:   M4Vec zerovec(0, 0, 0, 0);
 55:   for (std::size_t i = 0; i < 10; ++i) { lts.pfinal.push_back(zerovec); }
 56:   std::cout << "MParton:: [Constructor done]" << std::endl;
 57: }
 58:
 59: void MParton::Initialize() {
 60:   const std::vector<std::string> supported = {"yy_LUX", "yy_DZ"};
 61:   CID                                      = "P";
 62:   ProcPtr                                  = MSubProc(supported, CID);
 63: }
 64:
 65: // Destructor
 66: MParton::~MParton() {}
 67:
 68: // Initialize cut and process spesific postsetup
 69: void MParton::post_Constructor() {
 70:   if (ProcPtr.CHANNEL == "RES") {
 71:     // Here we support only single resonances
 72:     if (lts.RESONANCES.size() != 1) {
 73:       std::string str =
 74:           "MParton::post_Constructor: Only single resonance supported for this "
 75:           "process (RESPARAM.size() != 1)";
 76:       throw std::invalid_argument(str);
 77:     }
 78:   }
 79:
 80:   // Set sampling boundaries
 81:   SetTechnicalBoundaries(gcuts, EXCITATION);
 82:
 83:   // Initialize phase space dimension
```

./src/MParton.cc 2/4

```
 84:   ProcPtr.LIPSDIM = 2;  // All processes
 85:
 86:   // Not applicable here
 87:   // if (EXCITATION == 1) { ProcPtr.LIPSDIM += 1; }
 88:   // if (EXCITATION == 2) { ProcPtr.LIPSDIM += 2; }
 89: }
 90:
 91: // No screening loop kinematics considered here
 92: bool MParton::LoopKinematics(const std::vector<double> &plp, const std::vector<double> &p2p) {
 93:   return false;
 94: }
 95:
 96: // Return Monte Carlo integrand weight
 97: double MParton::EventWeight(const std::vector<double> &randvec, AuxIntData &aux) {
 98:   double W = 0.0;
 99:
100:   // Kinematics and cuts
101:   aux.kinematics_ok = B2RandomKin(randvec);
102:   aux.fidcuts_ok    = FiducialCuts();
103:   aux.vetocuts_ok   = VetoCuts();
104:
105:   if (aux.Valid()) {
106:     // Matrix element squared
107:     const double MatESQ = GetAmp2();
108:
109:     // Calculate central system Phase Space volume
110:     double exact = 0.0;
111:     DecayWidthPS(exact);
112:     lts.DW_sum_exact.Add(gra::kinematics::MCW(exact, pow2(exact), 1), aux.vegasweight);
113:
114:     // Add to the integral sum and take into account the VEGAS weight
115:     lts.DW_sum.Add(lts.DW, aux.vegasweight);
116:
117:     double C_space = 1.0;
118:     // We have some legs in the central system
119:     if (lts.decaytree.size() != 0 && lts.PS_active) {
120:       C_space = lts.DW.Integral();
121:
122:       // ------------------------------------------------------------------
123:       // Cascade resonances phase-space
124:       C_space *= CascadePS();
125:       // ------------------------------------------------------------------
126:     }
127:
128:     // ** EVENT WEIGHT **
129:     W = C_space * (1.0 / S_factor) * MatESQ * B2IntegralVolume() * B2PhaseSpaceWeight() * GeV2barn /
130:         MollerFlux();
131:   }
132:
133:   aux.amplitude_ok = CheckInfNan(W);
134:
135:   // As the last STEP: Histograms
136:   // if (!aux.burn_in_mode) {
137:   const double totalweight = W * aux.vegasweight;
138:   FillHistograms(totalweight, lts);
139:   //}
140:
141:   return W;
142: }
143:
144: // Fiducial cuts
145: bool MParton::FiducialCuts() const { return CommonCuts(); }
146:
147: // Record event
148: bool MParton::EventRecord(HepMC3::GenEvent &evt) { return CommonRecord(evt); }
149:
150: void MParton::PrintInit(bool silent) const {
151:   if (!silent) {
152:     PrintSetup();
153:
154:     // Construct prettyprint diagram
155:     std::string proton1 = "----------pdf-------->";
156:     std::string proton2 = "----------pdf-------->";
157:
158:     /*
159:     if (EXCITATION == 1) {
160:                         proton1 = "----------F2-xxxxxxxx>";
161:     }
162:     if (EXCITATION == 2) {
163:                         proton1 = "----------F2-xxxxxxxx>";
164:                         proton2 = "----------F2-xxxxxxxx>";
165:     }
166:     */
```

./src/MParton.cc 3/4

```
167:
168:     std::vector<std::string> feynmangraph;
169:     feynmangraph = {"||           ", "||           ", "xx--------->", "||           ", "||           "};
170:
171:     // Print diagram
172:     std::cout << proton1 << std::endl;
173:     for (const auto &i : indices(feynmangraph)) {
174:       if (SCREENING) {  // Put red
175:         std::cout << rang::fg::red << "     **    " << rang::style::reset;
176:       } else {
177:         std::cout << rang::fg::red << "           " << rang::style::reset;
178:       }
179:       std::cout << feynmangraph[i] << std::endl;
180:     }
181:     std::cout << proton2 << std::endl;
182:     std::cout << std::endl;
183:     std::cout << std::endl;
184:     std::cout << rang::style::bold << "Generation cuts:" << rang::style::reset << std::endl
185:               << std::endl;
186:
187:     std::cout << "- None (full x1,x2 range in use)" << std::endl;
188:
189:     PrintFiducialCuts();
190:   }
191: }
192:
193: // 2-dimensional phase space vector initialization
194: bool MParton::B2RandomKin(const std::vector<double> &randvec) {
195:   double x1 = randvec[0];
196:   double x2 = randvec[1];
197:
198:   const unsigned int MAXTRIAL = 1e4;
199:   unsigned int       trials   = 0;
200:   while (true) {
201:     double M_sum = 0.0;
202:
203:     // Pick daughter masses
204:     // ============================================================
205:     for (const auto &i : indices(lts.decaytree)) {
206:       GetOffShellMass(lts.decaytree[i], lts.decaytree[i].m_offshell);
207:       M_sum += lts.decaytree[i].m_offshell;
208:     }
209:     // ============================================================
210:
211:     if (x1 * x2 * lts.s > pow2(M_sum)) { break; }  // kinematics possible
212:
213:     ++trials;
214:     if (trials > MAXTRIAL) {
215:       return false;  // Impossible given daughter masses and this x1 and x2 value
216:     }
217:   }
218:   return B2BuildKin(x1, x2);
219: }
220:
221: // Build kinematics for 2 -> 1 x 1 -> N collinear
222: bool MParton::B2BuildKin(double x1, double x2) {
223:   static const M4Vec beamsum = lts.pbeam1 + lts.pbeam2;
224:
225:   // We-work in CMS-frame
226:
227:   // Initial state collinear (pt=0) and massless (m=0) parton 4-momentum
228:   M4Vec q1(0, 0, x1 * lts.sqrt_s / 2.0, x1 * lts.sqrt_s / 2.0);
229:   M4Vec q2(0, 0, -x2 * lts.sqrt_s / 2.0, x2 * lts.sqrt_s / 2.0);
230:
231:   // ------------------------------------------------------------------
232:   // Now boost if asymmetric beams
233:   if (std::abs(beamsum.Pz()) > 1e-6) {
234:     constexpr int sign = 1;  // positive -> boost to the lab
235:     kinematics::LorentzBoost(beamsum, lts.sqrt_s, q1, sign);
236:     kinematics::LorentzBoost(beamsum, lts.sqrt_s, q2, sign);
237:   }
238:   // ------------------------------------------------------------------
239:
240:   M4Vec p1 = lts.pbeam1 - q1;  // Remnant
241:   M4Vec p2 = lts.pbeam2 - q2;  // Remnant
242:   M4Vec pX = q1 + q2;          // System
243:
244:   // Save
245:   lts.pfinal[1] = p1;
246:   lts.pfinal[2] = p2;
247:   lts.pfinal[0] = pX;  // Central system
248:
249:   // ------------------------------------------------------------------
```

./src/MParton.cc 4/4

```
250:   // Kinematic checks
251:
252:   // Total 4-momentum conservation
253:   if (!CheckEMC(beamsum - (lts.pfinal[1] + lts.pfinal[2] + lts.pfinal[0]))) { return false; }
254:
255:   // ====================================================================
256:   // Central system decay tree first branch kinematics set up here, the
257:   // rest is done recursively
258:
259:   // Mother mass
260:   std::vector<double> masses;
261:
262:   // Collect decay product masses
263:   for (const auto &i : indices(lts.decaytree)) {
264:     // @@ Note, we need to take offshell masses here @@
265:     masses.push_back(lts.decaytree[i].m_offshell);
266:   }
267:   std::vector<M4Vec> products;
268:
269:   // false if amplitude has dependence on the final state legs (generic),
270:   // true if amplitude is a function of central system kinematics only (limited)
271:   const bool UNWEIGHT = !lts.PS_active;
272:
273:   gra::kinematics::MCW w;
274:   // 2-body
275:   if (lts.decaytree.size() == 2) {
276:     w = gra::kinematics::TwoBodyPhaseSpace(lts.pfinal[0], lts.pfinal[0].M(), masses, products,
277:                                            random);
278:     // 3-body
279:   } else if (lts.decaytree.size() == 3) {
280:     w = gra::kinematics::ThreeBodyPhaseSpace(lts.pfinal[0], lts.pfinal[0].M(), masses, products,
281:                                              UNWEIGHT, random);
282:     // N-body
283:   } else if (lts.decaytree.size() > 3) {
284:     w = gra::kinematics::NBodyPhaseSpace(lts.pfinal[0], lts.pfinal[0].M(), masses, products,
285:                                          UNWEIGHT, random);
286:   }
287:
288:   if (w.GetW() < 0) {
289:     return false;  // Kinematically impossible
290:   }
291:   lts.DW = w;
292:
293:   // Collect decay products
294:   const unsigned int offset = 3;
295:   for (const auto &i : indices(lts.decaytree)) {
296:     lts.decaytree[i].p4    = products[i];
297:     lts.pfinal[i + offset] = products[i];
298:   }
299:
300:   // Treat decaytree recursively
301:   for (const auto &i : indices(lts.decaytree)) {
302:     if (!ConstructDecayKinematics(lts.decaytree[i])) { return false; }
303:   }
304:
305:   // ====================================================================
306:   // Check that we are above mass threshold -> not necessary, this is
307:   // done in mass sampling function
308:   const unsigned int Nf = lts.decaytree.size() + 2;
309:   return GetLorentzScalars(Nf);
310: }
311:
312: // Calculate Pure Phase Space Decay Width (Volume)
313: void MParton::DecayWidthPS(double &exact) const {
314:   exact = 0;
315:   // Massive exact closed form for 2-body case
316:   if (lts.decaytree.size() == 2) {
317:     exact = gra::kinematics::PS2Massive(lts.m2, pow2(lts.decaytree[0].p4.M()),
318:                                         pow2(lts.decaytree[1].p4.M()));
319:   }
320:   // Massless case
321:   if (lts.decaytree.size() > 2) {
322:     exact = gra::kinematics::PSnMassless(lts.m2, lts.decaytree.size());
323:   }
324: }
325:
326: // 2-Dim Integral Volume
327: double MParton::B2IntegralVolume() const { return 1.0; }
328:
329: // 2-Dim phase space weight
330: double MParton::B2PhaseSpaceWeight() const { return 1.0; }
331:
332: }  // namespace gra
```

./src/MTransport.cc 1/3

```
  1: // Optimal Transport Algorithms
  2: //
  3: //
  4: // (c) 2017-2020 Mikael Mieskolainen
  5: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  6:
  7: // C++ standard
  8: #include <algorithm>
  9: #include <complex>
 10: #include <iostream>
 11: #include <random>
 12: #include <vector>
 13:
 14: // Own
 15: #include "Granitti/MMatOper.h"
 16: #include "Granitti/MMath.h"
 17: #include "Granitti/MMatrix.h"
 18: #include "Granitti/MTransport.h"
 19:
 20: // Eigen
 21: #include <Eigen/Dense>
 22:
 23: namespace gra {
 24: namespace opt {
 25:
 26:
 27: // Input:  n, m,    Kernel dimensions
 28: //         lambda,  Entropic regularization
 29: //
 30: // Output: K (n x m) Convolution kernel matrix
 31: //
 32: void ConvKernel(std::size_t n, std::size_t m, double lambda, MMatrix<double>& K) {
 33:   std::vector<double> x = math::linspace(0.0, n - 1.0, n);
 34:   matoper::ScaleVector(x, 1.0 / (n - 1.0));
 35:
 36:   std::vector<double> y = math::linspace(0.0, m - 1.0, m);
 37:   matoper::ScaleVector(y, 1.0 / (m - 1.0));
 38:
 39:   MMatrix<double> X;
 40:   MMatrix<double> Y;
 41:   matoper::MeshGrid(x, y, X, Y);
 42:
 43:   // Convolution matrix
 44:   K = MMatrix<double>(X.size_row(), X.size_col());
 45:   for (std::size_t i = 0; i < X.size_row(); ++i) {
 46:     for (std::size_t j = 0; j < X.size_col(); ++j) {
 47:       K[i][j] = std::exp(-math::pow2(X[i][j] - Y[i][j]) / lambda);
 48:     }
 49:   }
 50: }
 51:
 52: // Input:  lambda,    Entropic regularization
 53: //         C (n x m), Cost matrix
 54: //
 55: // Output: K (n x m) Gibbs (Gaussian convolution) kernel matrix
 56: //
 57: void GibbsKernel(double lambda, const MMatrix<double>& C, MMatrix<double>& K) {
 58:   // Gibbs Kernel: Entropy regularized distance matrix
 59:   K = MMatrix<double>(C.size_row(), C.size_col());
 60:   for (std::size_t i = 0; i < C.size_row(); ++i) {
 61:     for (std::size_t j = 0; j < C.size_col(); ++j) { K[i][j] = std::exp(-C[i][j] / lambda); }
 62:   }
 63: }
 64:
 65: // Sinkhorn-Knopp non-linear but convex optimization algorithm
 66: //
 67: // Input:
 68: //
 69: // Kernel matrix between elements of p and q              (n x m)
 70: // Probability density p (histogram / dirac point mass)   (n x 1)   with \sum = 1
 71: // Probability density q (histogram / dirac point mass)   (m x 1)   with \sum = 1
 72: // Iterations                                             (scalar)
 73: //
 74: // Output:
 75: //
 76: // Optimal Transport Matrix Pi                            (n x m)
 77: // Distance measure                                       (scalar)
 78: //
 79: double SinkHorn(MMatrix<double>& Pi, const MMatrix<double>& K, std::vector<double>& p,
 80:                 std::vector<double>& q, std::size_t iter) {
 81:   const double SAFE_EPS = 1e-64;
 82:
 83:   std::cout << "opt::SinkHorn optimization:" << std::endl;
```

./src/MTransport.cc 2/3

```
 84:
 85:   // ============================================================
 86:   // Test the normalization
 87:   auto fsum = [](const std::vector<double>& x) {
 88:     double value = 0.0;
 89:     for (std::size_t i = 0; i < x.size(); ++i) { value += x[i]; }
 90:     return value;
 91:   };
 92:
 93:   const double EPS = 1e-5;
 94:   if (std::abs(fsum(p) - 1) > EPS) {
 95:     const std::string str = "SinkHorn:: Input p elements sum != 1";
 96:     throw std::invalid_argument(str);
 97:   }
 98:   if (std::abs(fsum(q) - 1) > EPS) {
 99:     const std::string str = "SinkHorn:: Input q elements sum != 1";
100:     throw std::invalid_argument(str);
101:   }
102:   // ============================================================
103:
104:   const std::size_t n = K.size_row();
105:   const std::size_t m = K.size_col();
106:
107:   // Transposed version
108:   MMatrix<double> KT = K.Transpose();
109:
110:   // Initialization
111:   std::vector<double> u(n, 0.0);  // no need for initialization
112:   std::vector<double> v(m, 1.0);  // 1
113:
114:   // || Pi * 1 - p ||  or  || Pi^T 1 - q ||
115:   std::vector<double> ones_p(m, 1.0);
116:   std::vector<double> ones_q(n, 1.0);
117:
118:   auto resnorm_p = [&](const MMatrix<double>& Pi) {
119:     double                value = 0.0;
120:     const std::vector<double> lhs   = Pi * ones_p;
121:     for (std::size_t i = 0; i < lhs.size(); ++i) { value += std::abs(lhs[i] - p[i]); }
122:     return value;
123:   };
124:
125:   auto resnorm_q = [&](const MMatrix<double>& PiT) {
126:     double                value = 0.0;
127:     const std::vector<double> lhs   = PiT * ones_q;
128:     for (std::size_t i = 0; i < lhs.size(); ++i) { value += std::abs(lhs[i] - q[i]); }
129:     return value;
130:   };
131:
132:   // Sinkhorn iterations
133:   double p_cost = 0.0;
134:   double q_cost = 0.0;
135:
136:
137:   MMatrix<double> PiT;
138:
139:   for (std::size_t k = 0; k < iter; ++k) {
140:     const std::vector<double> Kv = K * v;  // >= 0
141:     for (std::size_t i = 0; i < n; ++i) { u[i] = p[i] / std::max(SAFE_EPS, Kv[i]); }
142:
143:     const std::vector<double> KTu = KT * u;  // >= 0
144:     for (std::size_t i = 0; i < m; ++i) { v[i] = q[i] / std::max(SAFE_EPS, KTu[i]); }
145:
146:     // Calculate metrics
147:     if ((k + 1) % (iter / 10) == 0 || k == 0 || k == iter - 1) {
148:       // Transport coupling matrix
149:       Pi  = matoper::diagAdiag(u, K, v);
150:       PiT = Pi.Transpose();
151:
152:       // Transport cost
153:       p_cost = resnorm_p(Pi);
154:       q_cost = resnorm_q(PiT);
155:
156:       printf(
157:           "iter = %4lu / %4lu : p_cost = %0.5E, log(p_cost) = %4.1f, q_cost = %0.5E, log(q_cost) = "
158:           "%4.1f \n",
159:           k + 1, iter, p_cost, std::log(p_cost), q_cost, std::log(q_cost));
160:     }
161:   }
162:
163:   return p_cost;
164: }
165:
166: }  // namespace opt
```

```
167: }  // namespace gra
168:
```



```
  1: // 'Durham QCD' Processes and Amplitudes
  2: //
  3: // (c) 2017-2020 Mikael Mieskolainen
  4: // Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  5:
  6: // C++
  7: #include <complex>
  8: #include <random>
  9: #include <vector>
 10:
 11: // Own
 12: #include "Graniitti/MAux.h"
 13: #include "Graniitti/MDurham.h"
 14: #include "Graniitti/MForm.h"
 15: #include "Graniitti/MKinematics.h"
 16: #include "Graniitti/MPDG.h"
 17: #include "Graniitti/MSudakov.h"
 18:
 19: // Libraries
 20: #include "json.hpp"
 21: #include "rang.hpp"
 22:
 23: namespace gra {
 24: using PDG::mp;
 25: using math::PI;
 26: using math::msqrt;
 27: using math::pow2;
 28: using math::zi;
 29:
 30: // 2-body incoming or outgoing helicity combinations
 31: // (keep it algebraic order to match with MadGraph)
 32: constexpr int MM = 0;  // --
 33: constexpr int MP = 1;  // -+
 34: constexpr int PM = 2;  // +-
 35: constexpr int PP = 3;  // ++
 36:
 37: // 2-body final state helicity configurations
 38: static const std::vector<int> fs2 = {MM, MP, PM, PP};
 39:
 40: // Central system J^P = 0^+, 0^-, +2^+, -2^+
 41: enum SPINPARITY { P0, M0, P2, M2 };  // Implicit conversion to int
 42:
 43: // Constructor
 44: MDurham::MDurham(gra::LORENTZSCALAR &lts, const std::string &modelfile) {
 45:   // @@ MULTITHREADING LOCK NEEDED FOR THE INITIALIZATION @@
 46:   gra::g_mutex.lock();
 47:
 48:   // Not created yet
 49:   if (lts.GlobalSudakovPtr == nullptr) { lts.GlobalSudakovPtr = new MSudakov(); }
 50:
 51:   // Not initialized yet
 52:   if (lts.GlobalSudakovPtr->initialized == false) {
 53:     try {
 54:       lts.GlobalSudakovPtr->Init(lts.sqrt_s, lts.LHAPDFSET, true);
 55:     } catch (...) {
 56:       gra::g_mutex.unlock();  // need to release here, otherwise get infinite lock
 57:       throw;
 58:     }
 59:   }
 60:   if (!Param.initialized) {
 61:     try {
 62:       Param.ReadParameters(modelfile);
 63:     } catch (...) {
 64:       gra::g_mutex.unlock();  // need to release here, otherwise get infinite lock
 65:       throw;
 66:     }
 67:   }
 68:   gra::g_mutex.unlock();
 69: }
 70:
 71: // Durham QCD / KMR model
 72: //
 73: // [REFERENCE: Pumplin, Phys.Rev.D 52 (1995)]
 74: // [REFERENCE: Khoze, Kaidalov, Martin, Ryskin, Stirling, arxiv.org/abs/hep-ph/0507040]
 75: // [REFERENCE: Khoze, Martin, Ryskin, arxiv.org/abs/hep-ph/0605113]
 76: // [REFERENCE: Harland-Lang, Khoze, Ryskin, Stirling, arxiv.org/abs/1005.0695]
 77: // [REFERENCE: Harland-Lang, Khoze, Ryskin, arxiv.org/abs/1409.4785]
 78: //
 79: double MDurham::DurhamQCD(gra::LORENTZSCALAR &lts, const std::string &process) {
 80:   // gluon pair continuum
 81:   if (process == "gg") {
 82:     // [Final state helicities/polarizations x 4 initial state helicities]
 83:     std::vector<std::vector<std::complex<double>>> Amp(4,
```



```
 84:                                                        std::vector<std::complex<double>>(4, 0.0));
 85:
 86:     // Madgraph
 87:     const double alpha_s = lts.GlobalSudakovPtr->AlphaS_Q2(lts.s_hat);
 88:     AmpMG5_gg_gg.CalcAmp2(lts, alpha_s);
 89:
 90:     // Amplitude evaluated outside the Qt-loop (approximation)
 91:     Dgg2gg(lts, Amp);
 92:
 93:     // Run loop
 94:     return DQtloop(lts, Amp);
 95:
 96:     // qqbar continuum
 97:   } else if (process == "qqbar") {
 98:     // [Final state helicities/polarizations x 4 initial state helicities]
 99:     std::vector<std::vector<std::complex<double>>> Amp(4,
100:                                                        std::vector<std::complex<double>>(4, 0.0));
101:
102:     /*
103:     // Madgraph
104:     const double alpha_s = lts.GlobalSudakovPtr->AlphaS_Q2(lts.s_hat);
105:     AmpMG5_gg_uubar.CalcAmp2(lts, alpha_s);
106:     */
107:
108:     // Amplitude evaluated outside the Qt-loop (approximation)
109:     Dgg2qqbar(lts, Amp);
110:
111:     // Run loop
112:     return DQtloop(lts, Amp);
113:
114:     // Meson pair continuum
115:   } else if (process == "MMbar") {
116:     // [Final state helicities/polarizations x 4 initial state helicities]
117:     std::vector<std::vector<std::complex<double>>> Amp(1,
118:                                                        std::vector<std::complex<double>>(4, 0.0));
119:
120:     // Amplitude evaluated outside the Qt-loop (approximation)
121:     Dgg2MMbar(lts, Amp);
122:
123:     // Run loop
124:     return DQtloop(lts, Amp);
125:
126:     // chic(0) resonance
127:   } else if (process == "chic(0)") {
128:     // [Final state helicities/polarizations x 4 initial state helicities]
129:     std::vector<std::vector<std::complex<double>>> Amp(1,
130:                                                        std::vector<std::complex<double>>(4, 0.0));
131:
132:     // Amplitude evaluated outside the Qt-loop (approximation)
133:     std::vector<double> null;
134:     Dgg2chic0(lts, Amp, null, null);
135:
136:     // Run loop
137:     return DQtloop(lts, Amp);
138:
139:     // Flux with |A| = 1
140:   } else if (process == "FLUX") {
141:     // [Final state helicities/polarizations x 4 initial state helicities]
142:     std::vector<std::vector<std::complex<double>>> Amp(1,
143:                                                        std::vector<std::complex<double>>(4, 0.0));
144:
145:     // Initial state gluon helicity combinations,
146:     // ++ and -- give contribution for 0+ state (see DHelProj function)
147:     const double A = 1.0;
148:
149:     std::vector<std::complex<double>> is(4, 0.0);
150:     is[MM] = A;
151:     is[MP] = 0;
152:     is[PM] = 0;
153:     is[PP] = A;
154:
155:     // No polarization for the final state to loop over, only one index
156:     Amp[0] = is;
157:
158:     // Run loop
159:     return DQtloop(lts, Amp);
160:
161:     // ------------------------------------------------------------------
162:     //
163:     // Implement more processes here ...
164:     //
165:     // ------------------------------------------------------------------
166:   } else {
```



```
167:     std::string str = "MDurham::Durham: Unknown subprocess: " + process;
168:     throw std::invalid_argument(str);
169:   }
170: }
171:
172: // Helicity basis decomposition
173: //
174: // In the forward limit q1_t = - q2_t = Q_t
175: // => gives gluon polarization vectors eps_1 = -eps_2 => central system J_z = 0
176: //
177: inline void MDurham::DHelicity(const std::vector<double> &q1, const std::vector<double> &q2,
178:                                std::vector<std::complex<double>> &JzP) const {
179:   const unsigned int X = 0;  // component for readability
180:   const unsigned int Y = 1;
181:
182:   // 1/2  q1_t dot q2_t
183:   JzP[P0] = -0.5 * (q1[X] * q2[X] + q1[Y] * q2[Y]);
184:
185:   // 1/2 i |q1_t x q2_t|
186:   JzP[M0] = -0.5 * math::zi * std::abs(q1[X] * q2[Y] - q1[Y] * q2[X]);
187:
188:   const double Re = 0.5 * (q1[X] * q2[X] - q1[Y] * q2[Y]);
189:   const double Im = 0.5 * (q1[X] * q2[Y] + q1[Y] * q2[X]);
190:
191:   // +1/2[ (xx - yy) + i(xy + yx) ]
192:   JzP[P2] = Re + math::zi * Im;
193:
194:   // +1/2[ (xx - yy) - i(xy + yx) ]
195:   JzP[M2] = Re - math::zi * Im;
196: }
197:
198: // M_{\lambda_1,\lambda_2}
199: // i.e. g(\lambda_1) g(\lambda_2) -> X (system) helicity amplitudes
200: //
201: // where lambda1, lambda2 are the initial state gluon helicities
202: //
203: // [REFERENCE: arxiv.org/abs/1405.0018v2, formula (20)]
204: //
205: inline std::complex<double> MDurham::DHelProj(const std::vector<std::complex<double>> &A,
206:                                               const std::vector<std::complex<double>> &JzP) const {
207:   // M_{++} + M_{--}
208:   // (this term gives J_z^PC = 0^++ selection rule in the forward pt->0 limit)
209:   const std::complex<double> aP0 = JzP[P0] * (A[PP] + A[MM]);
210:   // M_{++} - M_{--}
211:   const std::complex<double> aM0 = JzP[M0] * (A[PP] - A[MM]);
212:   // M_{-+}
213:   const std::complex<double> aP2 = JzP[P2] * A[MP];
214:   // M_{+-}
215:   const std::complex<double> aM2 = JzP[M2] * A[PM];
216:
217:   /*
218:   // DEBUG
219:   std::cout << "0+ : " << aP0 << std::endl;
220:   std::cout << "0- : " << aM0 << std::endl;
221:   std::cout << "2+ : " << aP2 << std::endl;
222:   std::cout << "2- : " << aM2 << std::endl;
223:   std::cout << std::endl << std::endl;
224:   */
225:
226:   // Coherent sum!
227:   return aP0 + aM0 + aP2 + aM2;
228: }
229:
230: /*
231: // MADGRAPH HELICITY FORMAT [direct algebraic order]
232: const static int helicities[ncomb][nexternal] =
233:     {{-1, -1, -1, -1}, 0
234:      {-1, -1, -1,  1}, 1
235:      {-1, -1,  1, -1}, 2
236:      {-1, -1,  1,  1}, 3
237:
238:      {-1,  1, -1, -1}, 4
239:      {-1,  1, -1,  1}, 5
240:      {-1,  1,  1, -1}, 6
241:      {-1,  1,  1,  1}, 7
242:
243:      { 1, -1, -1, -1}, 8
244:      { 1, -1, -1,  1}, 9
245:      { 1, -1,  1, -1}, 10
246:      { 1, -1,  1,  1}, 11
247:
248:      { 1,  1, -1, -1}, 12
249:      { 1,  1, -1,  1}, 13
```

```
250:    { 1,  1,  1, -1}, 14
251:    { 1,  1,  1,  1}}; 15
252: */
253:
254: // Alternative (semi-ad-hoc) scenarios for the scale choise
255: inline void MDurham::DScaleChoise(double qt2, double q1_2, double q2_2, double &Q1_2_scale,
256:                                   double &Q2_2_scale) const {
257:   if (Param.PDF_scale == "MIN") {
258:     Q1_2_scale = std::min(qt2, q1_2);
259:     Q2_2_scale = std::min(qt2, q2_2);
260:   } else if (Param.PDF_scale == "MAX") {
261:     Q1_2_scale = std::max(qt2, q1_2);
262:     Q2_2_scale = std::max(qt2, q2_2);
263:   } else if (Param.PDF_scale == "IN") {
264:     Q1_2_scale = q1_2;
265:     Q2_2_scale = q2_2;
266:   } else if (Param.PDF_scale == "EX") {
267:     Q1_2_scale = qt2;
268:     Q2_2_scale = qt2;
269:   } else if (Param.PDF_scale == "AVG") {
270:     Q1_2_scale = (qt2 + q1_2) / 2.0;
271:     Q2_2_scale = (qt2 + q2_2) / 2.0;
272:   } else {
273:     throw std::invalid_argument("MDurham::DScaleChoise: Unknown 'Durham::PDF_scale' option!");
274:   }
275: }
276: //
277: // [REFERENCE: Khoze, Martin, Ryskin, journals.aps.org/prd/pdf/10.1103/PhysRevD.56.5867]
278: // [REFERENCE: Harland-Lang, Khoze, Martin, Ryskin, Stirling, arxiv.org/abs/1405.0018v2]
279: //
280: // See also:
281: // [REFERENCE: Lonnblad, Zlebcik, arxiv.org/abs/1608.03765]
282: //
283: //
284: // Durham loop integral amplitude:
285: // A = pi^2 \int \frac{d^2 Q_t M(gg->X)}{Q_t^2(Q_t-(p_1t))^2(Q_t+p_{2t})^2}
286: //                 x f_g(x_1,x_1',Q_1^2,\mu_2;t_1) x
287: //                 f_g(x_2,x_2',Q_2^2,\mu_2;t_2)
288: //
289: double MDurham::DQtloop(gra::LORENTZSCALAR &                                  lts,
290:                         std::vector<std::vector<std::complex<double>>> Amp) {
291:   // Forward (proton) system pt-vectors
292:   const std::vector<double> pt1 = {lts.pfinal[1].Px(), lts.pfinal[1].Py()};
293:   const std::vector<double> pt2 = {lts.pfinal[2].Px(), lts.pfinal[2].Py()};
294:
295:   // ***********************************************************************
296:   // ** Process scale (GeV) / Sudakov suppression kt^2 integral upper bound **
297:   const double MU = msqrt(lts.s_hat / Param.alphas_scale);
298:   // ***********************************************************************
299:
300:   // Init 2D-Simpson weight matrix (will be calculated only once, being
301:   // static), C++11 handles multithreaded static initialization
302:   const static MMatrix<double> WSimpson = math::Simpson38Weight2D(Param.N_qt, Param.N_phi);
303:
304:   // NOTE N + 1, init with zero!
305:
306:   // Amp.size == 4 for example for the gg -> gg process [initial states summed coherently]
307:   std::vector<MMatrix<std::complex<double>>> f(
308:       Amp.size(), MMatrix<std::complex<double>>(Param.N_qt + 1, Param.N_phi + 1, 0.0));
309:
310:   // Spin-Parity
311:   std::vector<std::complex<double>> JzP(4, 0.0);
312:
313:   // 2D-loop integral
314:   //
315:   // \int d^2 \vec{qt} [...] = \int dphi \int dqt qt [...]
316:   //
317:
318:   // Linearly discretised qt-loop, N+1!
319:   for (std::size_t i = 0; i < Param.N_qt + 1; ++i) {
320:     const double qt  = Param.qt_MIN + i * Param.qt_STEP;
321:     const double qt2 = pow2(qt);
322:
323:     // Linearly discretized phi in [0,2pi), N+1!
324:     for (std::size_t j = 0; j < Param.N_phi + 1; ++j) {
325:       const double qphi = j * Param.phi_STEP;
326:
327:       // ---------------------------------------------------------------------
328:       // Loop vector
329:       const std::vector<double> qt_ = {qt * std::cos(qphi), qt * std::sin(qphi)};
330:
331:       // Fusing gluon pt-vectors
332:       const std::vector<double> q1 = {qt_[0] - pt1[0], qt_[1] - pt1[1]};
```



```
333:       const std::vector<double> q2 = {qt_[0] + pt2[0], qt_[1] + pt2[1]};
334:
335:       const double q1_2 = math::vpow2(q1);
336:       const double q2_2 = math::vpow2(q2);
337:
338:       // Get fusing gluon spin-parity (J_z^P) components
339:       // {g1,g2} -> {0^+,0^-,+2^+,-2^-}
340:       DHelicity(q1, q2, JzP);
341:
342:       // ** Durham scale choise **
343:       double Q1_2_scale = 0.0;
344:       double Q2_2_scale = 0.0;
345:       DScaleChoise(qt2, q1_2, q2_2, Q1_2_scale, Q2_2_scale);
346:
347:       // Minimum scale cutoff
348:       if (Q1_2_scale < Param.qt2_MIN || Q2_2_scale < Param.qt2_MIN) { continue; }
349:
350:       // -----------------------------------------------------------------------
351:       // Get amplitude level pdfs
352:       const double fg_1 = lts.GlobalSudakovPtr->fg_xQ2M(lts.x1, Q1_2_scale, MU);
353:       const double fg_2 = lts.GlobalSudakovPtr->fg_xQ2M(lts.x2, Q2_2_scale, MU);
354:
355:       // Amplitude weight:
356:       // * \pi^2 : see original KMR papers: [\alpha_s CF -> pi x f_g] for fg_1 and fg_2
357:       // *   2  : factor from initial state boson-statistics, check it:!
358:       // *   qt : jacobian of d^2qt -> dphi dqt qt
359:       std::complex<double> weight = fg_1 * fg_2 / (qt2 * q1_2 * q2_2);
360:       weight *= math::PIPI * 2.0 * qt;
361:
362:       // Loop over (outgoing) helicity combinations.
363:       // Amp[h] contains initial state gluon helicity combinations --,-+,+-,++
364:       // Here we sum coherently
365:       for (std::size_t h = 0; h < f.size(); ++h) { f[h](i)(j) = weight * DHelProj(Amp[h], JzP); }
366:
367:     }  // phi-loop
368:   }    // |q_t|-loop
369:
370:   // Evaluate the total numerical integral for each helicity amplitude
371:   std::vector<std::complex<double>> sum(f.size(), 0.0);
372:   for (std::size_t h = 0; h < f.size(); ++h) {
373:     sum[h] = math::Simpson38Integral2D(f[h], WSimpson, Param.qt_STEP, Param.phi_STEP);
374:   }
375:
376:   // *****Make sure it is of right size!*****
377:   lts.hamp.resize(f.size());
378:
379:   // Outgoing helicity combinations
380:   double A2 = 0.0;
381:   for (std::size_t h = 0; h < f.size(); ++h) {
382:     lts.hamp[h] = sum[h];
383:
384:     // Apply proton form factors
385:     lts.hamp[h] *=
386:         lts.excite1 ? gra::form::S3FINEL(lts.t1, lts.pfinal[1].M2()) : gra::form::S3F(lts.t1);
387:     lts.hamp[h] *=
388:         lts.excite2 ? gra::form::S3FINEL(lts.t2, lts.pfinal[2].M2()) : gra::form::S3F(lts.t2);
389:
390:     // Apply phase space factors
391:     lts.hamp[h] *= msqrt(16.0 * math::PIPI);
392:     lts.hamp[h] *= msqrt(16.0 * math::PIPI);
393:     lts.hamp[h] *= msqrt(lts.s / lts.s_hat);
394:
395:     // For the total amplitude squared (for no eikonal screening applied)
396:     A2 += math::abs2(lts.hamp[h]);
397:   }
398:   // Initial state helicity average 1/4 not needed here
399:
400:   // -----------------------------------------------------------------------
401:   // Amplitude cutoff (hard perturbative limit)
402:   if (gra::math::msqrt(lts.m2) < 2.0) {
403:     A2      = 0.0;
404:     lts.hamp = std::vector<std::complex<double>>(f.size(), 0.0);
405:   }
406:   // -----------------------------------------------------------------------
407:
408:   return A2;
409: }
410:
411: // ===========================================================================
412: // Exclusive \chi_c(0+) sub-amplitude
413: //
414: // [REFERENCE: Khoze, Martin, Ryskin, Stirling, arxiv.org/abs/hep-ph/0403218]
415: // [REFERENCE: Pasechnik, Szczurek, Teryaev, arxiv.org/abs/0709.0857v1]
```



```
416: //
417: void MDurham::Dgg2chic0(const gra::LORENTZSCALAR &                            lts,
418:                         std::vector<std::vector<std::complex<double>>> &Amp,
419:                         const std::vector<double> &qt1, const std::vector<double> &qt2) const {
420:   const double alpha_s = lts.GlobalSudakovPtr->AlphaS_Q2(lts.s_hat / Param.alphas_scale);
421:
422:   const double gs2   = 4.0 * PI * alpha_s;  // coupling
423:   const double K_NLO = 1.68;                // NLO correction
424:   const double M0    = 3.41475;             // chi_c(0+) mass (GeV)
425:   const double W0    = 0.0108;              // chi_c(0+) width (GeV)
426:   const double NC    = 3.0;                 // #colors
427:
428:   // Gluonic width \Gamma(\chi_c(0+) -> gg), see references
429:   std::complex<double> A = K_NLO * 8.0 * math::zi * gs2 / M0 * msqrt(0.075) / msqrt(PI * M0 * NC);
430:
431:   // Apply Breit-Wigner propagator shape delta-function
432:   A *= gra::form::deltaBWamp(lts.s_hat, M0, W0);
433:
434:   // Initial state gluon helicity combinations,
435:   // ++ and -- give contribution for 0+ state (see DHelProj function)
436:   std::vector<std::complex<double>> is(4, 0.0);
437:   is[MM] = A;
438:   is[MP] = 0;
439:   is[PM] = 0;
440:   is[PP] = A;
441:
442:   // No polarization for the final state to loop over, only one index
443:   Amp[0] = is;
444: }
445:
446: // ===========================================================================
447: // gg -> gg tree-level helicity amplitudes
448: //
449: //
450: // Basic result: d\hat{\sigma}/dt = 9/4 \pi \alpha_s^2 / E_T^4
451: //
452: // In pure gluon amplitudes: when gluon helicities are the same,
453: // or at most one is different from the rest, vanish for any n >= 4,
454: // where n is the total number of gluons (in+out)
455: //
456: // [REFERENCE: Dixon, arxiv.org/abs/1310.5353v1]
457: //
458: // Note that incoming gluons (a,b) are in a color singlet state,
459: // i.e., a = b or \delta^{ab} Tr [ .color algebra. ] has been applied
460: //
461: void MDurham::Dgg2gg(const gra::LORENTZSCALAR &                                lts,
462:                      std::vector<std::vector<std::complex<double>>> &Amp) {
463:   const double alpha_s = lts.GlobalSudakovPtr->AlphaS_Q2(lts.s_hat / Param.alphas_scale);
464:
465:   // Vertex factor coupling, gs^2 = 4 pi alpha_s
466:   double norm = 4.0 * PI * alpha_s;
467:
468:   // Color part
469:   const double NC = 3.0;
470:   norm *= NC / msqrt(NC * NC - 1);
471:   norm *= 2.0;
472:
473:   // Helicity phase
474:   const double             phi      = lts.decaytree[0].p4.Phi();
475:   const std::complex<double> negphase = std::exp(-zi * phi);
476:   const std::complex<double> posphase = std::exp(zi * phi);
477:
478:   const double aux01 = pow2(lts.s_hat) / (lts.u_hat * lts.t_hat);
479:   const double aux23 = lts.u_hat / lts.t_hat;
480:
481:   // We use natural binary order / Madgraph order here
482:   std::vector<std::complex<double>> dualAmp(16, 0.0);
483:
484:   // Loop over final state gluon pair helicity combinations
485:   for (std::size_t h = 0; h < fs2.size(); ++h) {
486:     std::vector<std::complex<double>> is(4, 0.0);
487:     if (fs2[h] == PP || fs2[h] == MM) {
488:       is[MM] = (fs2[h] == PP) ? 0 : norm * aux01;  // -- => ++ or --
489:       is[MP] = 0;                                  // -+ => ++ or --
490:       is[PM] = 0;                                  // +- => ++ or --
491:       is[PP] = (fs2[h] == PP) ? norm * aux01 : 0;  // ++ => ++ or --
492:     } else if (fs2[h] == PM || fs2[h] == MP) {
493:       is[MM] = 0;                                                  // -- => +- or -+
494:       is[MP] = negphase * norm * ((fs2[h] == PM) ? 1 / aux23 : aux23);  // -+ => +- or -+
495:       is[PM] = posphase * norm * ((fs2[h] == PM) ? aux23 : 1 / aux23);  // +- => +- or -+
496:       is[PP] = 0;                                                  // ++ => +- or -+
497:     }
498:     Amp[h] = is;
```



```
499:
500:     for (std::size_t k = 0; k < 4; ++k) { dualAmp[4 * k + h] = is[k]; }
501:   }
502:
503:   // -------------------------------------------------------------------------
504:   const bool DEBUG = false;
505:   if (DEBUG) {
506:     // TEST amplitude squared WITH MADGRAPH (UNDER IMPLEMENTATION)
507:     double madsum  = 0.0;
508:     double thissum = 0.0;
509:     for (unsigned int i = 0; i < 16; ++i) {
510:       printf("i=%2d , |mad|^2 = %0.3f , |amp|^2 = %0.3f \n", i, math::abs2(lts.hamp[i]),
511:              math::abs2(dualAmp[i]));
512:       madsum += math::abs2(lts.hamp[i]);
513:       thissum += math::abs2(dualAmp[i]);
514:     }
515:     Asum += thissum / madsum;
516:     Nsum += 1.0;
517:     printf("THIS/MADGRAPH = %0.10f \n\n", Asum / Nsum);
518:   }
519:   // -------------------------------------------------------------------------
520: }
521:
522: // ===========================================================================
523: // gg -> meson pair (gg -> q\bar{q} q\bar{q})
524: //
525: // [REFERENCE: Harland-Lang, Khoze, Ryskin, Stirling, arxiv.org/pdf/1105.1626.pdf]
526:
527: // Meson wave-function (without scale Q&\201)dependence)
528: // x the longitudinal momentum fraction of a parton within the meson
529: // fM the meson decay constant
530: //
531: // Normalization: \int_0^1 dx phi(x) \equiv fM/(2\sqrt{3})
532: //
533: double MDurham::phi_CZ(double x, double fM) const {
534:   return 5.0 * std::sqrt(3.0) * fM * x * (1.0 - x) * pow2(1.0 - 2.0 * x);
535:   // return 6.0*std::sqrt(3.0)*fM * x*(1.0 - x);
536: }
537:
538: // For tabulating the meson wave function
539: //
540: // [REFERENCE: Takizawa,
541: // http://www-nucleth.phys.sci.osaka-u.ac.jp/jp-usw/talk/Takizawa.pdf]
542: //
543: // Here, one could perhaps update the phenomenology (check more recent papers).
544: //
545: // -------------------------------------------------------------------------
546: // Three charge neutral state are in the SU(3)_F quark model nonet
547: // (octet+singlet):
548: // pi0, eta8, eta0
549: //
550: // Rotation (mixing) to physical basis via:
551: //
552: // |eta > = |cos th  -sin th| |eta8>
553: // |eta'>   |sin th   cos th| |eta0>
554: //
555: // Decay constants:
556: //
557: // f_{8eta}  =  f_8 * cos th
558: // f_{8eta'} =  f_8 * sin th
559: //
560: // f_{0eta}  = -f_0 * sin th
561: // f_{0eta'} =  f_0 * cos th
562: //
563: std::vector<double> MDurham::EvalPhi(int N, int pdg) const {
564:   // Meson decay constants
565:   double                 fM       = 0.0;
566:   static const std::vector<double> supported = {111, 211, 321, 311};  // pi0, pi+, K+, K0
567:   if (std::find(supported.begin(), supported.end(), pdg) != supported.end()) {
568:     fM = PDG::fM_meson.at(pdg);
569:   } else {
570:     fM = PDG::fM_meson.at(111);  // else, take pi0
571:   }
572:   const double fM_eta8 = fM * 1.0;  // just use pi0
573:   const double fM_eta0 = fM * 1.0;  // just use pi0
574:
575:   // Mixing angle
576:   static const double THETA_eta = math::Deg2Rad(-15.4);
577:   const double        costh     = std::cos(THETA_eta);
578:   const double        sinth     = std::sin(THETA_eta);
579:
580:   // Evaluate
581:   std::vector<double> f(N + 1);
```

```
582:   const double        STEP = 1.0 / N;
583:   for (const auto &i : aux::indices(f)) {
584:     const double x = i * STEP;
585:     if (pdg == 221) {  // eta via mixing
586:       f[i] = costh * phi_CZ(x, fM_eta8) - sinth * phi_CZ(x, fM_eta0);
587:     } else if (pdg == 331) {  // etaprime via mixing
588:       f[i] = sinth * phi_CZ(x, fM_eta8) + costh * phi_CZ(x, fM_eta0);
589:     } else {
590:       f[i] = phi_CZ(x, fM);
591:     }
592:   }
593:   return f;
594: }
595:
596: // Meson pair amplitude
597: //
598: void MDurham::Dgg2MMbar(const gra::LORENTZSCALAR &                          lts,
599:                         std::vector<std::vector<std::complex<double>>> &Amp) {
600:   // Wave function x-discretization
601:   // [Easy speed & accuracy improvement: Check lambda-functions below,
602:   // they could be pre-calculated and interpolated as a function of costheta]
603:   const int     Nx      = 96;
604:   const double STEPx = 1.0 / Nx;
605:
606:   // Init 2D-Simpson weight matrix (will be calculated only once, being
607:   // static), C++11 handles multithreaded static initialization
608:   const static MMatrix<double> WSimpson = math::Simpson38Weight2D(Nx, Nx);
609:
610:   const double delta_AB = 8;  // Sum over \delta^AB [gluons in with the same color]
611:   const double NC       = 3;  // Three colors
612:   const double CF       = (pow2(NC) - 1.0) / (NC * 2.0);  // SU(3) algebra
613:
614:   const double alpha_s = lts.GlobalSudakovPtr->AlphaS_Q2(lts.s_hat / Param.alphas_scale);
615:
616:   const double shat      = lts.s_hat;
617:   const double m         = lts.decaytree[0].p4.M();
618:   const double beta      = kinematics::Beta(pow2(m), shat);
619:   const double costheta  = (1.0 + 2.0 * lts.t_hat / lts.s_hat - 2.0 * pow2(m) / lts.s_hat) / beta;
620:   const double costheta2 = pow2(costheta);
621:
622:   // ------------------------------------------------------------------
623:   // ** Hard angular cut-off **
624:   // some sub-amplitudes are singular when |costheta| -> 1
625:
626:   if (std::abs(costheta) > Param.MAXCOS) {
627:     Amp[0] = std::vector<std::complex<double>>(4, 0.0);  // Return zero
628:     return;
629:   }
630:   // ------------------------------------------------------------------
631:
632:   // Helicity phase
633:   const double                phi       = lts.decaytree[0].p4.Phi();
634:   const std::complex<double> posphase = std::exp(2.0 * zi * phi);
635:   const std::complex<double> negphase = std::exp(-2.0 * zi * phi);
636:
637:   // ------------------------------------------------------------------
638:   // Mesons scalar flavor octet (non-singlet) amplitude:
639:   // |\pi0\pi0>, |\pi+\pi->, |K+K->, |K0\bar{K0}>
640:   //
641:   // T_+- = T_-+
642:   auto T_SFO_PM = [&](double x, double y) {
643:
644:     const double a = (1.0 - x) * (1.0 - y) + x * y;  // +
645:     const double b = (1.0 - x) * (1.0 - y) - x * y;  // -
646:
647:     return 1.0 / (x * y * (1.0 - x) * (1.0 - y)) * (x * (1.0 - x) + y * (1.0 - y)) /
648:            (pow2(a) - pow2(b) * costheta2) * (NC / 2.0) * (costheta2 - 2.0 * CF / NC * a);
649:   };
650:   // ------------------------------------------------------------------
651:
652:   // ------------------------------------------------------------------
653:   // SU(3)_F scalar flavor-singlet amplitude:
654:   // |\eta'\eta'>
655:   // |\eta\eta>, |\eta\eta'> (|\eta-\eta'> mixing)
656:   //
657:   // T_++ = T_--
658:   auto T_SFS_PP = [&](double x, double y) {
659:
660:     return 1.0 / (x * y * (1.0 - x) * (1.0 - y)) * (1.0 + costheta2) / pow2(1.0 - costheta2);
661:   };
662:
663:   // T_+- = T_-+
664:   auto T_SFS_PM = [&](double x, double y) {
```



```
665:
666:     return 1.0 / (x * y * (1.0 - x) * (1.0 - y)) * (1.0 + 3.0 * costheta2) /
667:            (2.0 * pow2(1.0 - costheta2));
668:   };
669:   // ------------------------------------------------------------------
670:
671:   // pi0, pi+, K+, K0
672:   static const std::vector<int> SFO_PDG = {111, 211, 321, 311};
673:   // eta, eta'
674:   static const std::vector<int> SFS_PDG = {221, 331};
675:
676:   static const int pdg0 = std::abs(lts.decaytree[0].p.pdg);
677:   static const int pdg1 = std::abs(lts.decaytree[1].p.pdg);
678:
679:   // Evaluate only once the meson wave functions
680:   static const std::vector<double> wfphi0 = EvalPhi(Nx, pdg0);
681:   static const std::vector<double> wfphi1 = EvalPhi(Nx, pdg1);
682:
683:   // ------------------------------------------------------------------
684:   const double                CUTOFF = 1e-15;  // To avoid singularity at x = 0, x = 1
685:   static const std::vector<double> xval   = math::linspace(CUTOFF, 1.0 - CUTOFF, Nx + 1);
686:
687:   // ------------------------------------------------------------------
688:   // Integral over meson wave functions:
689:   // M\int_0^1 dx dy \phi_M(x) \phi_\bar{M}(y) T_{\lambda\lambda'}
690:   // (x,y,\hat{s},\theta)
691:
692:   // Normalization factor
693:   const double norm = (delta_AB / NC) * (64.0 * math::PIPI * pow2(alpha_s)) / shat;
694:
695:   // Scalar flavor octet
696:   if ((std::find(SFO_PDG.begin(), SFO_PDG.end(), pdg0) != SFO_PDG.end())) {
697:     std::vector<std::complex<double>> is(4);
698:
699:     MMatrix<double> f_PM(Nx + 1, Nx + 1, 0.0);
700:
701:     for (std::size_t i = 0; i < Nx + 1; ++i) {
702:       for (std::size_t j = 0; j < Nx + 1; ++j) {
703:         f_PM[i][j] = wfphi0[i] * wfphi1[j] * T_SFO_PM(xval[i], xval[j]);
704:       }
705:     }
706:
707:     is[PP] = 0.0;
708:     is[MM] = 0.0;
709:     is[PM] = norm * math::Simpson38Integral2D(f_PM, WSimpson, STEPx, STEPx) * posphase;
710:     is[MP] = is[PM] * negphase;
711:
712:     Amp[0] = is;
713:   }
714:
715:   // Scalar flavor singlet
716:   else if ((std::find(SFS_PDG.begin(), SFS_PDG.end(), pdg0) != SFS_PDG.end())) {
717:     std::vector<std::complex<double>> is(4);
718:
719:     MMatrix<double> f_PP(Nx + 1, Nx + 1, 0.0);
720:     MMatrix<double> f_PM(Nx + 1, Nx + 1, 0.0);
721:
722:     for (std::size_t i = 0; i < Nx + 1; ++i) {
723:       for (std::size_t j = 0; j < Nx + 1; ++j) {
724:         f_PP[i][j] = wfphi0[i] * wfphi1[j] * T_SFS_PP(xval[i], xval[j]);
725:         f_PM[i][j] = wfphi0[i] * wfphi1[j] * T_SFS_PM(xval[i], xval[j]);
726:       }
727:     }
728:
729:     is[PP] = norm * math::Simpson38Integral2D(f_PP, WSimpson, STEPx, STEPx);
730:     is[MM] = is[PP];
731:     is[PM] = norm * math::Simpson38Integral2D(f_PM, WSimpson, STEPx, STEPx) * posphase;
732:     is[MP] = is[PM] * negphase;
733:
734:     Amp[0] = is;
735:   } else {
736:     throw std::invalid_argument("MDurham::Dgg2MMbar: Unsupported meson: " +
737:                                 std::to_string(lts.decaytree[0].p.pdg));
738:   }
739: }
740:
741: // ===================================================================
742: // gg -> qqbar tree-level helicity amplitudes
743: //
744: //
745: void MDurham::Dgg2qqbar(const gra::LORENTZSCALAR &                          lts,
746:                         std::vector<std::vector<std::complex<double>>> &Amp) {
747:   throw std::invalid_argument("MDurham::DurhamQCD: qqbar amplitude in the next version");
```



```
748: }
749:
750: }  // namespace gra
751:
```

ISBN 978-951-51-5942-7

Helsinki 2020