Nuclear Instruments and Methods in Physics Research A 1062 (2024) 169150

Contents lists available at ScienceDirect

Nuclear Inst. and Methods in Physics Research, A

journal homepage: www.elsevier.com/locate/nima

Full Length Article

PyPWA: A software toolkit for parameter optimization and amplitude analysis

Mark Jones [a], Peter Hurck [b,*], William Phelps [c,d], Carlos W. Salgado [a,d]

[a] Norfolk State University, Norfolk, 23504, VA, USA
[b] University of Glasgow, Glasgow, G12 8QQ, United Kingdom
[c] Christopher Newport University, Newport News, 23606, VA, USA
[d] The Thomas Jefferson National Accelerator Facility, Newport News, 23606, VA, USA

ARTICLE INFO

Keywords:
Amplitude analysis
Optimization
Python
Hadron spectroscopy
Data analysis
Partial wave analysis

ABSTRACT

PyPWA is a toolkit designed to optimize parametric models describing data and generate simulated distributions according to a model. Its software has been written within the python ecosystem with the goal of performing Amplitude or Partial Wave Analysis (PWA) in nuclear and particle physics experiments. We briefly describe the general features of amplitude analysis and we provide a description of the PyPWA software design and usage. We also provide benchmarks of the scaling and an example of its application.

1. Introduction

1.1. A brief description of amplitude analysis

In particle and nuclear physics, one of the main experimental goals is to determine the frequency with which a particular interaction occurs. Experimental data are analyzed under the theoretical framework of quantum field theories. The data can be presented as counting histograms, in different bins of the particles' kinematic variables. The counting probability is represented by differential cross-sections that can be written, using Fermi's golden rule [1], by

$$\frac{d\sigma}{d\tau} \propto \sum_{ext.\ spins} \int |\mathscr{M}(\tau)|^2 dx^n \tag{1}$$

where $\mathscr{M}$, the transition amplitude, describes the physics (dynamics) of the particular interaction. The integral is done over all the out-going particle's four-momenta (phase-space) taking into account kinematics constraints (represented by dx^n). The transition amplitude is a complex function that cannot be measured directly; it has to be inferred from the cross-sections. In most cases, the amplitudes (therefore, cross-sections) depend on several kinematic variables, represented here by τ (e.g. beam energy, the total mass of the final state set of particles, the angular distribution of the final particles, etc.). The experimentally measured kinematic variables are related to the intrinsic quantum numbers involved in the reaction. The main goal of the analysis is to extract those quantum numbers and, through them, gain information about the fundamental components of the reaction. Of special importance are the angular momentum and spin quantum numbers, which are related to the angular distributions of the decay products.

Different techniques are used to extract information about the properties of amplitudes by measuring cross-sections or quantities related to cross-sections. Dalitz-Plot analysis [2] is used to study correlations between the kinematic variables involved in the reaction. These kinematic distributions are fit with a theoretically motivated parametric representation of the amplitudes. In most amplitude studies, it is common to consider only the transition amplitude ($\mathscr{M}$), and by binning the data in one or more of the kinematic variables τ', a subset of τ (to suppress the dependence on the binned variable), such that an *intensity*, I, is defined by

$$I(\tau') = \sum_{ext.\ spins} |\mathscr{M}(\tau')|^2. \tag{2}$$

The transition amplitudes $\mathscr{M}$ are defined by the model or theory. When the intensity is only dependent on angular variables, we can expand it in Harmonic Moments (Moments method) [3], or we can write the spin components using the Spin Density Matrix, the intensity is then parameterized by the matrix elements (SDME method) [4]. Alternatively, we can expand the intensity in Partial Waves defined by the angular components (PWA method) [5,6]. These approaches are mathematically related and have their advantages and disadvantages. The PWA method directly obtains intensities classified by their angular quantum numbers and is more general but also more complex.

* Corresponding author.
E-mail address: Hurck@glasgow.ac.uk (P. Hurck).

https://doi.org/10.1016/j.nima.2024.169150
Received 12 June 2023; Received in revised form 2 February 2024; Accepted 2 February 2024
Available online 4 March 2024
0168-9002/© 2024 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

All of these methods, in practice, are reduced to the optimization of amplitude model parameters to describe the data. Therefore, any software infrastructure used for these types of analyses has to perform the parameter optimization (from now on referred to as fitting) and the generation of simulated data.

In the following we will describe how the PyPWA toolkit aims to fill those necessities by providing the user with modular software that will facilitate all tasks required for an amplitude analysis. One example of PWA using PyPWA for the reaction $\gamma p \to pX \to p\eta\pi$ is presented in Appendix A. Further examples and tutorials are included with the software and documentation on the PyPWA website [7].

2. Toolkit requirements

2.1. Simulation, fitting, and prediction

The toolkit's main tasks are generating simulated events, optimizing parametric models, and generating simulated data based on the optimized model (from here on referred to as prediction). The generation of simulated data by PyPWA is performed by two methods. In the first method, data are simulated through rejection sampling from a model with a priori parameters (resonances masses, widths, ...), and since the model contains all the dynamics, it is performed on unbinned data. A rejection mask is applied to the full set of previously generated Lorentz phase space Monte Carlo. This simulation can be performed to improve the understanding of the analysis or to design an experiment. A second method is used to simulate or predict the properties of the data according to the optimized (fitted) model parameters, allowing the predicted kinematic properties to be compared to the data. This method also uses rejection sampling but is applied to binned data, as the calculated parameters are normally given as a function of binned variables. Additionally, this last method can be folded with detector acceptances incorporated through an independent detector simulation, e.g. Geant4 [8].

One way to optimize the model, implemented in the software, is the maximization of the extended likelihood that is defined as [5]

$$\mathscr{L} = \left(\frac{\mathscr{N}^N}{N!} e^{-\mathscr{N}} \right) \prod_i^N I(\tau_i', \vec{V}) \tag{3}$$

where N is the number of data events and $\vec{V}$ is the set of model parameters. The total number of expected events, $\mathscr{N}$, calculated using events from a detector simulation, is defined as

$$\mathscr{N} = \eta \frac{1}{N_a} \sum_j^{N_a} I(\tau_j', \vec{V}) \tag{4}$$

where η is the detector acceptance and N_a is the number of accepted simulated events. For numerical reasons and because most available optimizers minimize a loss function, it is more efficient to use the negative logarithm of the likelihood. To obtain the optimal $\vec{V}$ values, the following is minimized

$$-ln\mathscr{L} = -\sum_{i=1}^N lnI(\tau_i', \vec{V}) + \mathscr{N}. \tag{5}$$

Optimization can be done with a variety of packages available within PyPWA. We provide an interface to iMinuit [9,10], and emcee [11] for cases where a Markov Chain Monte Carlo (MCMC) method is preferred (see Section 3.6).

2.2. Other packages

Prior to the development of PyPWA other photoproduction PWA packages existed, such as PWA2000 [12]. These packages were predominantly written in C/C++ and inspired PyPWA to develop in Python and use a modular structure for improved user experience. Some other examples that were developed both before and while PyPWA was developed include ComPWA [13], AmpTools [14], GPUPWA [15], and ROOTPWA [16]. The ComPWA project started with C++ and recently developed into a Python-based toolkit that provides a modular and flexible package for PWA. AmpTools is a C++ toolkit focusing on automation and leverages CUDA [17] for GPU access.

3. Software design and implementation

PyPWA was designed to be a flexible set of tools within the Python ecosystem to fit multi-dimensional models and generate simulations. It uses an object-oriented design for data structures and components, that are stored at runtime or support a plugin-like design.

PyPWA is a package built from individual and mostly independent components, which yields the flexibility of a toolkit design, allowing for use beyond its original scope. PyPWA has two categories for its components: data processing and data analysis. For data processing, PyPWA contains its own set of libraries for parsing, masking, and operating directly on data from multiple different data types. For data analysis, there are tools to aid in handling likelihoods, fitting, simulation, data splitting, and data visualization.

3.1. Choice of language

Python was selected because it already had the necessary tools and libraries in its ecosystem dedicated to numerical processing and analysis, such as NumPy [18], SciPy [19], and Matplotlib [20]. In addition, Python provides support for binding to other languages with tools such as F2PY [21] for Fortran or Cython [22] for C/C++ which enable the user to introduce a model in a different language. As an interpreted language, Python allows for interactive development. Two tools support interactive development: iPython [23] and Jupyter notebooks [24]. The iPython interpreter is similar to ROOT's CLING C/C++ interpreter [25] and is executed from the command line. Jupyter notebooks are web browser-based, have rearrangeable code cells that can be executed in any order, and provide markdown blocks for inline documentation. Matplotlib plots and standard output of the Python code are displayed below the code cell in which they are written. One limitation of Python is that it requires a Global Interpreter Lock (GIL), a mutex in the CPython implementation of Python that synchronizes access to Python objects, preventing multiple native threads from executing Python bytecodes concurrently. To bypass the GIL in PyPWA, the Multiprocessing module uses forks to parallel the execution of Python code. Multiprocessing is a key feature of PyPWA discussed in Section 3.5. Python can take advantage of libraries compiled in lower-level languages, e.g., C and Fortran, which are capable of low-level optimization, such as using vector instructions. NumPy, PyTorch [26], Matplotlib, and iMinuit are examples of the optimized libraries used. NumPy and SciPy allow for the development of amplitudes that PyPWA can automatically scale to all available hardware threads.

To increase the speed of the fits, there are three ways to boost performance: Multithreading, vectorized instructions, and general-purpose computing on GPUs (GPGPU). Modern CPUs have more than one core, with eight cores being common, and this is the first place in which performance can be increased by using all available cores with multiprocessing. Some architectures, such as the x86 architecture, have vector instructions as a part of the CPU instruction set, also known as "Single Instruction/Multiple Data" (SIMD), which allows the processor to perform multiple floating point operations in a single clock cycle. NumPy has built-in support for vector instructions, allowing users to write vectorized code in Python seamlessly. The final way to increase the performance of PyPWA is to offload the computations to GPGPU. For GPGPU calculations, the PyTorch Tensor Library supports AMD and NVIDIA GPGPUs and Apple's Silicon, allowing for a broad range of supported hardware. Without these libraries, users would be required to write their code in C/C++ or Fortran to achieve similar performance levels.

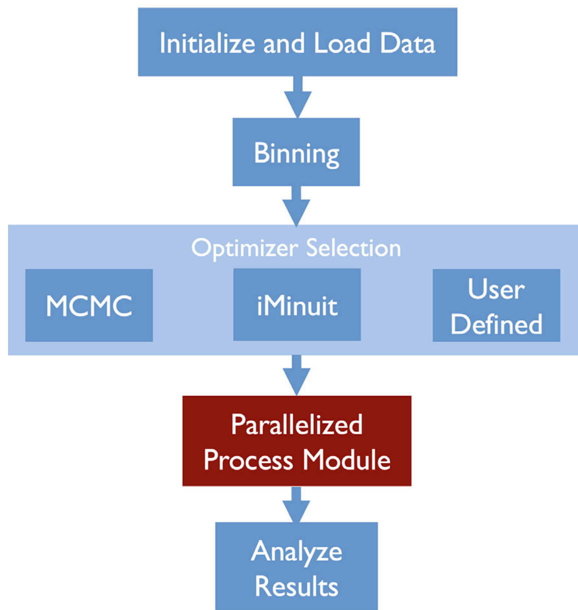

Fig. 1. Process flow control diagram for PyPWA shows the steps from loading to analyzing the data. The parallelized process module is described in Fig. 2.

3.2. Data preparation

Fig. 1 outlines the entire fitting process with PyPWA. After the user perform the initial event selection, data can be loaded in the following formats: GAMP [27], CSV, NumPy, or user-defined format representing a 2-dimensional array or a tree of 4-vectors. The data module leverages the plugin architecture for built-in and user-defined formats to implement the data caching described below. The amplitudes do not interact directly with the data module. The data module reads and writes the data from a Numpy Structured Array or a Pandas DataFrame. Those data structures are then passed from module to module, and the Processing Module, Data Module, Binning Module, etc., are all written to understand those data structures. When data is loaded with the data module, it will return a Structured Array, unless the user requested Pandas DataFrame. Then, when the data structure is sent to one of the other tools in the toolkit, it will know how to interpret and manipulate it. While the PyPWA data module is the preferred, most compatible way to handle data, it is not the only way. The user has the flexibility of not using the PyPWA data module and can swap it for Numpy, Pandas, or some other data parsing module such as Uproot [28], as long as what it returns is a Numpy Structured Array, DataFrame, or can be changed into one of those two types.

The default loading and writing mechanisms have additional features not found in the other tools, specifically plugins and caching. The data module inside PyPWA supports plugins that can define readers and writers, parsers, and dumpers. A reader and writer will read and write a single event to and from a file, whereas a parser and dumper will read the entire file into a structured array or write an entire array out to a file. While most operations use a parser or a dumper, the reader and writer can feed events through tools to be processed and immediately written back out to a file without consuming significant resources. This mechanism is used by the data masking tool built into the package. By supporting plugins, we can allow other developers to define file formats that best suit their needs and adapt PyPWA's parser to their file format while leveraging the rest of the toolkit. Lastly, any file written or read with PyPWA's data module is cached using Python's pickle module. Inside the cache is a SHA512 [29] sum of the original file, and if the sum does not match the file's current sum, the cache is invalidated, parsing the source file again.

The data module will return all parsed data in NumPy structured arrays or Panda's DataFrames. DataFrames are similar to structured arrays but have additional methods that allow direct data visualization and syntax for filtering and searching. NumPy structured arrays focus on numerical computing performance and, as such, have no tools for data visualization. We provide utilities to swap between these two data structures to allow users the best of both tools.

Finally, many users may need to split datasets into subsets, called bins, in one or more variables, e.g., bins of energy or mass. PyPWA supports multiple binning functions out of the box, allowing data to be split by a given range or by a fixed number of events. For both functions, the user provides a dataset to be binned, and the function will return a list of bins. If binning by range, the bins returned will all contain the same widths in the data, with the remaining data included in the first and last bins, decided using a single specified variable from that dataset. When binning by a fixed number of events, each bin will have the same number of events, again with the remaining data included in the first and last bins.

3.3. Software workflow

PyPWA provides a set of tools that help the user at each step of the analysis. A collection of the most important PyPWA functions and methods is shown in Table 1 (see Section 3.4 for more details). In this section we will describe how a possible workflow will be constructed for a so-called mass independent PWA (events are binned in mass to suppress the dynamical dependence on the mass, and an independent fit is performed for the events in each bin).

The user should prepare, in advance, the data and a simulation. The type of simulation needed will depend on the type of analysis. For example, in a full PWA using an extended likelihood, just a Lorentz invariant restricted phase-space of the reaction is needed. It is not required, but it is highly recommended to prepare the data and simulation information in a compressed version that contains only the variables used by the amplitude. This will simplify and speed up the reading of data using the `PyPWA.read()` method. PyPWA provides several functions in the binning module to organize the data and simulation in bins (in multi-dimensions). Examples of how to transform the data into more efficient inputs to the amplitudes and binning are provided in our tutorials. For example, the `PyPWA.bin_by_range()` will bin the data in N bins for a given variable range.

The next step is to define the intensity based on the model the user wants to use for the analysis. The user needs to define the `setup` and `calculate` functions. The following is an example using a two-dimensional Gaussian model of intensity:

```
class Gauss2dAmplitude(pwa.NestedFunction):
    """
    This is an example with a simple 2D Gaussian intensity. It is written to use PyPWA's built-in
    multiprocessing module. The user doesn't need to worry about thread or
    process management, passing data between threads, or any other
    hassles that come with multithreading.

    Instead, the user defines the class while extending the NestedFunction,
    and when it is passed to the fitter, it will clone the user's
    class, split the user's data, and deploy to every processing thread the
    user's hardware has.
    """

    def __init__(self):
        """
        The user can override the init function if they need to set parameters
        before the amplitude is passed to the likelihood or simulation
        functions. An example of this is included in the other tutorials.
        However, if this is done, the user must always remember to call the `super` function.
        """
        super(Gauss2dAmplitude, self).__init__()

    def setup(self, array):
        """
        This function is where the data is passed. Here, the user can also
        load any C or Fortran external libraries that typically would not
        support being packaged in Python's pickles (built-in serialization).
        """
        self.__x = array["x"]
        self.__y = array["y"]
```

Table 1
Summary of the most important methods and functions in PyPWA grouped by their functionality (see Section 3.4 for more details). PyPWA.NestedFunction() is the abstract base class used to define an amplitude inside the PyPWA framework, for this reason, the base class is used for both simulation and fitting.

Data processing	Simulation	Fitting
PyPWA.read()	PyPWA.monte_carlo_simulation()	PyPWA.ChiSquared()
PyPWA.write()	PyPWA.simulate.process_user_function()	PyPWA.LogLikelihood()
PyPWA.panda_to_numpy()	PyPWA.simulate.make_rejection_list()	PyPWA.EmptyLikelihood()
PyPWA.to_contiguous()	PyPWA.NestedFunction()	PyPWA.minuit()
PyPWA.bin_by_range()		PyPWA.mcmc()
PyPWA.bin_by_fixed_width()		PyPWA.NestedFunction()

```
def calculate(self, params):
    """
    This function receives the parameters from the optimizer and
    returns the values from there. Only the intensity values should
    be calculated here. The likelihood will be calculated elsewhere.
    """
    scaling = 1 / (params["A2"] * params["A4"])
    left = ((self.__x - params["A1"])**2)/(params["A2"]**2)
    right = ((self.__y - params["A3"])**2)/(params["A4"]**2)
    return scaling * np.exp(-(left + right))
```

After the intensity is defined, PyPWA provides methods to calculate the loss function for the optimizer. An example is shown below:

```
"""
Initial parameters for the optimizer.
"""
fitting_settings = {
    "A1": 1, "A2": 1,
    "A3": 1, "A4": 1,
}
"""
Optimization by Maximum Likelihood.
"""
for one_bin in binned_data:
    with pwa.LogLikelihood(
        Gauss2dAmplitude(), one_bin
    ) as likelihood:
        optimizer = pwa.minuit(fitting_settings, likelihood)

        # Set the fitting boundaries for the optimizer
        for param in ["A1", "A3"]:
            optimizer.limits[param] = (.1, None)

        for param in ["A2", "A4"]:
            optimizer.limits[param] = (1, None)

        # Record the final values from Minuit
        cpu_final_values.append(optimizer.migrad())
```

In the example, `PyPWA.LogLikelihood` calculates the loss function (log-likelihood in this case), and `PyPWA.minuit` defines the optimizer (Minuit in this case). Other methods exist in PyPWA to define other loss functions and optimizers (as described in Section 3.4).

In addition, PyPWA also contains modules and methods for simulation. `PyPWA.simulate` is normally used to produce a sample of data according to a model.

The `PyPWA.monte_carlo_simulation()` method is used to generate a boolean mask of accepted events (using the rejection sampling method) from an input Lorentz-phase space simulation. To check the quality of the fit, the user can use the `PyPWA.simulate` module also for prediction, based on a detector simulation, using the fitted parameters. The prediction step is normally performed on binned data, as the fitted amplitude values are normally calculated on a bin basis. We refer to the tutorial on our website [7] for code examples.

3.4. Functions, methods, and classes

PyPWA follows a Python ecosystem structure, providing modules that are imported by the default initialization. These modules contain classes (blueprints for creating objects), methods (functions associated with and called from an object), and functions (standalone blocks of code that perform a specific task). This subsection identifies the most important methods, functions, and classes that PyPWA provides. Full documentation is provided online (see Section 3.7). Furthermore, if the user runs PyPWA interactively, e.g., using Jupyter Lab notebook or PyCharm, there is access to the documentation by just typing `pwa?` or the function/method's name followed by a question mark.

The data handling is done mainly by two functions: `PyPWA.read(...)` that reads the entire file and returns either DataFrame (Pandas), ParticlePool (PyPWA's own 4-momenta format), or standard Numpy array. The data can be cached to speed up future use.

`PyPWA.write(...)` writes the entire file and returns either DataFrame (Pandas), ParticlePool, or standard Numpy array. For both read and write, the user can opt out of the default caching mechanism or forcefully invalidate and remove the cache.

Several loss functions to be used by the optimizers can be defined. They need to be defined by extending an abstract class. These loss functions automatically distribute the fitting function across available resources. Intensities can be defined using either an object-oriented (OOP) approach (class) or a functional programming approach (function).

If using pure functions for the intensity, users wrap the calculation function and optional setup methods in `PyPWA.FunctionalAmplitude(...)`, if using the OOP approach, they extend the `PyPWA.NestedFunction` class when defining the intensity. It is assumed by both, the fitting and simulation, that the `calculate` function/method of either approach will return a PyTorch Tensor, a Pandas Series, or a Numpy Array. They are expected to be initialized when sent to the likelihood or simulation objects and will be deep-copied for each process. The `setup` will be called first to initialize data and additional libraries, and then the `calculate` function/method will be called for each call to the likelihood. There are several switches that the user can set to obtain more efficient use of the hardware (described in the online documentation).

PyPWA supports two distinct likelihood types for use with the optimizer (see Section 3.6): The class `PyPWA.LogLikelihood(...)` defines the likelihood and works with either the standard or the extended log-likelihood, `PyPWA.sweightedLogLikelihood(...)` provides a log-likelihood that functions much like the above likelihood but supports weighted data (i.e. using sPlot [30]).

The class `PyPWA.ChiSquared(...)` computes the Chi-Squared loss function for a given amplitude. This loss function supports two types of the Chi-squared, one with binned and one with unbinned values. The class `PyPWA.EmptyLikelihood(...)` does no post operation on the final values except sum the array and return the final sum. This allows for defining unique loss functions that have not been defined by default, fitting functions that do not require a likelihood, or using the built-in multi-processing without the weight of a standard likelihood.

Optimizers are described in more detail in Section 3.6. They can be accessed through the following two functions:
`PyPWA.minuit(settings, likelihood)` optimization using iminuit and `PyPWA.mcmc(...)` optimization using emcee.

Simulation of data is performed through the following functions:

`PyPWA.simulate.process_user_function(...)` produces an array of values from a given function, `PyPWA.simulate.make_rejection_list(intensities, max_value)` produces the rejection list from pre-calculated function values (using the rejection method).

To organize and change data formats the function `PyPWA.pandas_to_numpy(...)` takes a Pandas Series or DataFrame and converts it to Numpy. Pandas have a built-in "*to_records*" function. However, records are slower than Structured Arrays, while containing much of the same functionality. `PyPWA.to_contiguous(data, names)` takes a dataset and a list of column names and converts those columns into contiguous arrays. The reason to use contiguous arrays over DataFrames or Structured arrays is that the memory is better aligned to improve computation speed. However, this doubles the memory requirements of the dataset since this copies all the events to the new array.

In many cases, the data analysis requires grouping the data into small sub-sets ("binning"). Binning of data can be done using the function `PyPWA.bin_with_fixed_widths (DataFrame, ...)` that bins data by fixed bin widths using a series in memory. The user specifies a width for all bins. Before use, all data that the user wants to be binned needs to be put into a DataFrame or Structured Array. Each resulting bin can be further binned if desired. If the `fixed_size` does not evenly divide into the length of `bin_series`, the first and last bin will contain overflows. The function `PyPWA.bin_by_range(dataframe, ...)` bins data by range, specifying the number of bins in that range, using a series in memory. Before use, all data that is about to be binned must be put into the DataFrame or Structured Array. Each resulting bin can be further binned if desired. `PyPWA.bin_by_list(data, bin_series, bin_list)` bins data using a list of bin widths (each bin can have a different width).

3.5. Parallel processing module

The Global Interpreter Lock (GIL) mechanism hinders Python's parallel processing ability by restricting the parallel execution of Python on one interpreter. As a way to bypass the GIL, multiprocessing is used to enable multiple interpreters to run concurrently. However, multiprocessing, which uses forks, presents technical challenges since each process operates independently without shared memory, requiring inter-process communication through system pipes. Specifically, two pipes per process are necessary to enable bi-directional communication.

PyPWA implements multiprocessing by inheriting from the Process class in the Multiprocessing module. The features that are added include duplex/simplex communication and kernels. In PyPWA, a kernel refers to a unit of execution that contains user-defined amplitude and its assigned data subset. The processing module is a process factory that takes a kernel and dataset and then scales those across several processes, defaulting to the total number of hardware threads on the host system. These kernels should not be confused with the user amplitude (NestedFunction). When fed to either a likelihood or to the simulation, the amplitude is then placed in a kernel defined by that module, which is then handed to the process factory. For example, suppose the LogLikelihood is selected for fitting. In that case, the amplitude will be placed inside the LogLikelihoodKernel, which will call the amplitude and use its result to compute the likelihood before returning it to the main thread. The processing factory will duplicate the provided kernel for each number, N, of selected hardware threads, split the dataset into N sub-datasets, and then attach each sub-dataset into one duplicated kernel as shown in Fig. 2. The kernels are assigned to a child process or thread with their data payloads. After the processes are forked, the processing factory returns a communication object that behaves like a function that passes values to all processes using pipes and processes the resulting values. The user primarily interacts with the processing module in the form of the likelihood module and simulation module.

Multiprocessing with forks works for many computational workloads; however, some libraries, such as PyTorch and CUDA, are incompatible. To provide multi-GPU support, threads are leveraged instead of multiprocessing forks. Even though a single GIL binds the threads and cannot execute more than one at a time, with GPU-heavy workloads, the majority of the threads' time is spent waiting on the GPUs to return a result, and due to this scales similarly as seen with CPUs, as discussed in Section 4.

The processing module combined with the instruction-level vectorization in NumPy and PyTorch allows for high scalability across hardware resources. This scalability is built directly into PyPWA and requires no additional developmental effort from the user.

3.6. Optimizer implementation

As introduced in Section 2 a loss function, such as a negative log-likelihood function, is minimized to extract model parameters that best fit the data. PyPWA requires the user to provide the intensity function $I(\tau', \vec{V})$ (*c.f.* Eq. (2)) which is used when fitting the data. This should be done in the form of a Python class that extends the NestedFunction and defines at least two methods: `setup` and `calculate`. The first will handle the data input and initialization, while the second will perform the actual calculation and return the intensity as a function of the parameters. PyPWA will use this Python class in the likelihood object it passes to an optimizer.

In general, an optimizer is required to be able to accept the loss function provided by PyPWA as input together with the set of parameters. It should then be able to return the final set of parameters that optimize the loss function together with their associated uncertainties. Ideally, one would like to be able to also constrain the range of individual parameters in the optimization. It should be possible to use any optimizer that has this capability within PyPWA.

PyPWA provides two built-in optimizers which are described in the following. Due to its modular nature, it is possible to add additional optimizers.

3.6.1. iMinuit

A common tool to perform fits to data in high-energy physics is MINUIT2 (based on [10]). PyPWA utilizes the Python implementation iMinuit [9].

MINUIT2 itself is not a minimizer but provides a range of algorithms that perform the minimization. The most common choice is MIGRAD, which uses the first and second derivatives of the function to be minimized in order to find the best set of parameters and approximate uncertainties. In a follow-up step, the user can use the HESSE or MINOS algorithms to improve the estimation of parameter uncertainties. For more information on the exact working and best use cases for each algorithm, consult Ref. [10] or the MINUIT2 User's Guide [31].

PyPWA provides an interface that takes the user-defined intensity function, i.e. the Python class, and input data and automatically turns it into a loss function which is passed to iMinuit. See Section 3.3 for an example on how to use the interface.

iMinuit provides some useful utility methods to inspect the final results. It is recommended that the user should carefully inspect the set of best parameters and uncertainties provided by iMinuit. It should be noted that HESSE and MINOS packages provide a better uncertainty estimation than MIGRAD and careful study of the uncertainties may be required.

3.6.2. emcee — Markov Chain Monte Carlo

Instead of using a minimizer, PyPWA offers the user the option to perform parameter estimation via Markov Chain Monte Carlo (MCMC). For this, the Python package *emcee* [11] is used.

In contrast to a minimization of a loss function, MCMC explores the possible parameter space. A range of different algorithms to do this

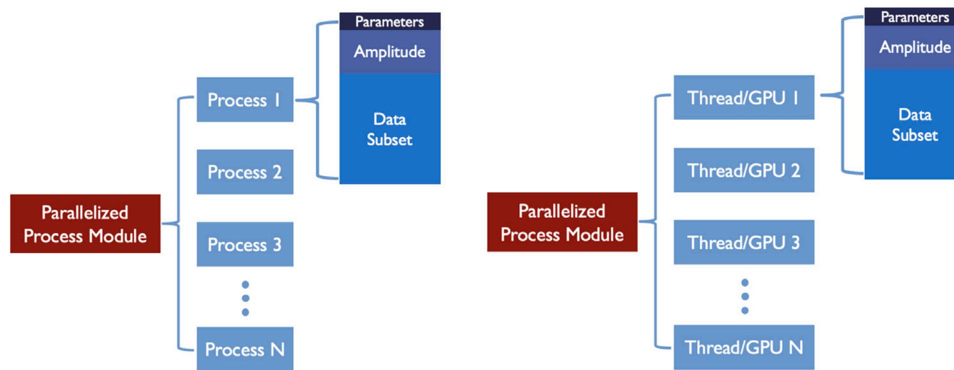

Fig. 2. The parallel processing module in PyPWA leverages multiprocessing to perform CPU-bound tasks concurrently, enabling scaling across all available hardware threads while bypassing the limitations of the GIL. Multi-GPU fitting uses multithreading instead of multiprocessing; one core in this diagram would correspond to one GPU. To the left, the default process uses forks and CPUs with a GIL per process; to the right, the threaded-based processing for GPUs bound to a single GIL.

is provided by *emcee*. In general, they can be split into the following stages:

1. Choose a set of starting parameters $\vec{a}_0$
2. propose a new set of parameters $\vec{a}_{n+1}$ (step)
3. calculate likelihood
4. accept or reject the proposed step based on likelihood
   if accepted: add $\vec{a}_{n+1}$ to output chain
   if rejected: add $\vec{a}_n$ to output chain
5. go back to 2.

Repeat this for a certain number of steps n.

The proposal of a new set of parameters $\vec{a}_{n+1}$ is often done at random based on the previous step $\vec{a}_n$. One possibility is to choose $\vec{a}_{n+1} = \vec{a}_n + \mathcal{N}(0, \vec{\sigma})$, where each parameter gets modified by a random number drawn from a normal distribution $\mathcal{N}$. The set of widths $\vec{\sigma}$ which determine how much each parameter is changed at every step is called step size. Whether a new step is accepted depends on the likelihood of the current (n) and new step ($n+1$). It is not advisable to always accept steps that increase the likelihood as this might cause the algorithm to get stuck in local maxima. In order to avoid local maxima, steps that decrease the likelihood should be accepted with a certain probability.

PyPWA provides the interface to access *emcee's* MCMC sampler via the `mcmc` module. Several different algorithms (moves) which propose and accept steps are included. The main strength of *emcee* is in so-called *ensemble sampling* in which multiple chains are run in parallel. For the full documentation of all moves, Ref. [11] should be consulted. PyPWA returns the `EnsembleSampler` object which can be used to retrieve all the available information from *emcee*. Most importantly, it can be used to get the resulting Markov chain, the complete history of all steps. This chain can be examined to study correlations between parameters and to extract a set of parameters that best describe the data together with their uncertainties. A useful tool to investigate the resulting chain is the so-called *corner plot* which can be created using the *corner* package [32]. An example for such a corner plot is shown in Fig. 3. It shows a sub-sample of the parameters used in the example in Appendix A. The parameter values for all individual steps in the chain are plotted against each other and reveal correlations. The corner plot also produces the one-dimensional projections which are often useful.

When using MCMC to perform parameter estimation it is important to study the generated chains carefully. It is necessary to ensure that the chains run long enough to converge. Although a step only depends on its immediate predecessor, there will be some auto-correlation within the chains, because steps are usually small enough to keep the acceptance rate reasonably high. This auto-correlation needs to be accounted for when information is to be extracted from the chain based on statistically independent samples. A common way to achieve this is to *thin out* the chain. That means that, depending on the auto-correlation length, e.g., only one in every 10, 20, or 50 steps of the chain is used for parameter estimation.

3.7. Documentation

In PyPWA, the documentation is contained within the source code allowing Python's docstrings to grow and mature with the code leveraging PEP 256 [33] and PEP 287 [34], which define Python's included support for literate programming. The docstring format used in PyPWA was initially defined for NumPy and blends an easily parsed markdown format for documentation generators while remaining syntactically simple to aid reading prior to rendering.

Python's built-in help function lets users view the docstrings attached to modules, classes, and functions directly from an IDE, iPython or Jupyter, which provides an interactive, built-in manual to PyPWA. For users and developers there is another tool inside Python's ecosystem, *Sphinx*, that can parse the docstrings and render the documentation into a static website, which *Read the Docs* currently hosts for PyPWA.

Documentation can be found on the PyPWA *Read the Docs* [35] and the website [7]. The full source code can be found on the GitHub page [36].

4. Benchmarks

The benchmarks presented in this work evaluate the scaling of PyPWA fits on multi-core CPUs and multiple GPUs. The benchmarks were performed on a dual-socket Intel Cascade Lake system with three NVIDIA V100 GPUs. A detailed overview of the system specifications can be found in Appendix B. The benchmark only considers the calculation of the loss function as it is typically the most computationally expensive part of the fit. The data loading and visualization tasks are one-time tasks that typically have a negligible effect on the runtime. The optimizer can have a variable number of function calls to the loss function, which necessitates the exclusion of these tasks to provide uniform benchmark conditions.

The benchmark procedure involves recording the execution time for a loss function call based on Appendix A, which is intended to mimic a fit, that uses a predetermined sequence of simulated fit parameters that are cached for the selected benchmark amplitude. This synthetic benchmark has 11 million events for both simulated experimental data and accepted data in the loss function and the number of loss function calls was set to 2636 calls, which had previously been observed in a fit performed with the default optimizer. This approach ensures that the benchmark has uniform conditions for all runs. The only parameter that varies is the number of threads utilized for the loss function calculation. Nonetheless, there exist uncontrolled factors in the benchmark testing

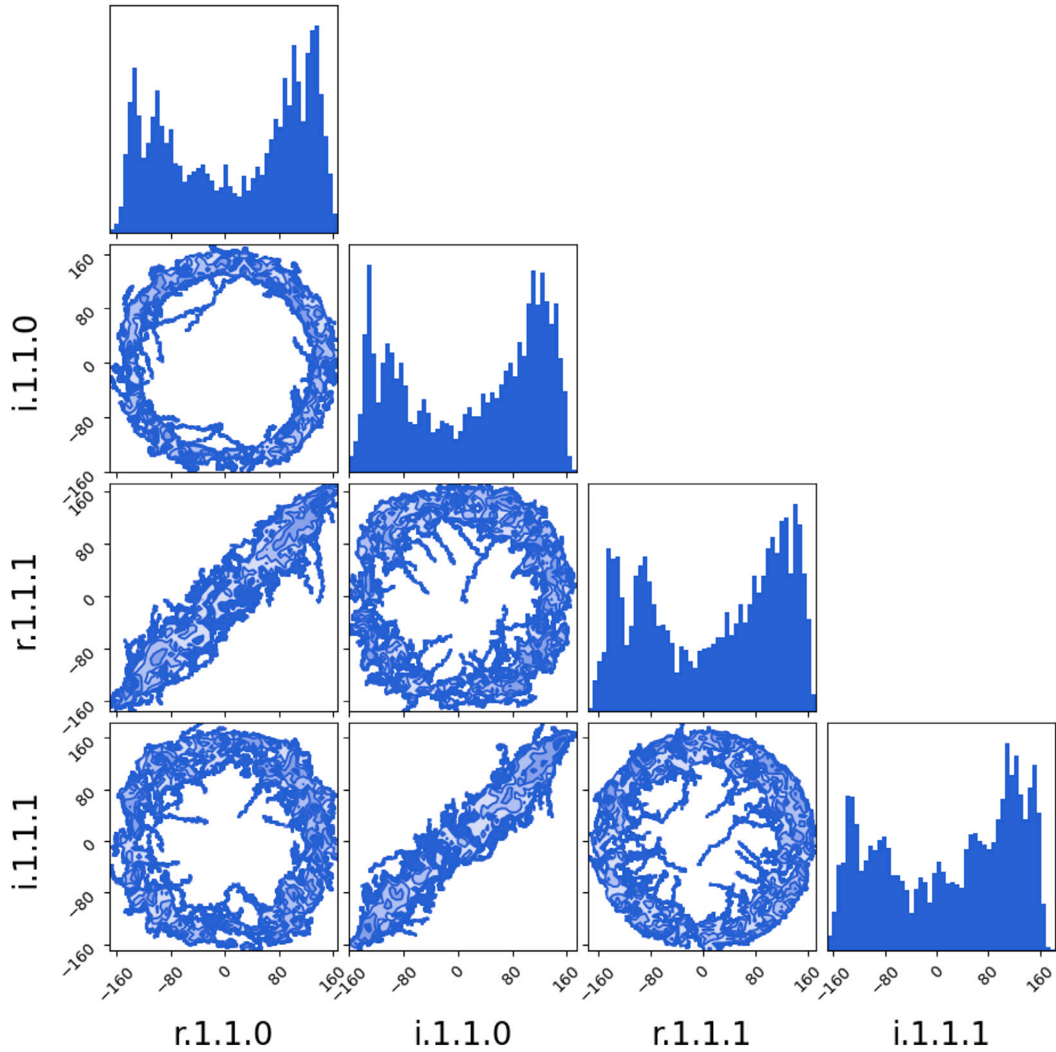

Fig. 3. Corner plot for a sub-sample of the parameters used in the example shown in Appendix A. The axes show the real (r) and imaginary (i) part of the production amplitudes abbreviated by their quantum numbers ϵ, L, and m in arbitrary units. The plot nicely visualizes correlations between the parameters. It was produced using the *corner* package [32].

Table 2
Results for the benchmarks as described in the text. The execution time is specified for 80 threads (or three GPUs in the case of the V100). Also listed are the speedups S as defined in the text and ratio p as defined in Eq. (7).

Math library	Execution time (in s)	S	p
NumPy	1447.25 ± 2.48	15.51 ± 0.03	94.74%
NumExpr	1268.83 ± 2.14	12.40 ± 0.02	93.10%
NumExpr ST	1180.56 ± 2.02	18.19 ± 0.03	95.70%
PyTorch: 3 V100s	80.70 ± 2.16	2.92 ± 0.08	98.61%

Table 3
Comparison of execution times for a single thread and 80 threads using various math libraries described in the text. In the case of the V100 only one CPU thread was used.

Math library	Execution time one process (in s)	Execution time 80 processes (in s)
NumPy	$22\,440.85 \pm 13.13$	1447.25 ± 2.48
NumExpr	$15\,731.39 \pm 115.31$	1268.83 ± 2.14
NumExpr ST	$21\,473.77 \pm 46.79$	1180.56 ± 2.02
PyTorch: 1 V100	235.53 ± 5.81	–
PyTorch: 3 V100s	80.70 ± 2.16	–

process due to the variability in the server environment and variable CPU clock frequency which is known as Intel Turbo Boost for Intel processors that implement this feature.

The benchmark system had two Intel CPUs with forty threads each and three NVIDIA V100 GPUs and is described in Table B.5. In order to measure the scaling there was one benchmark configuration for each thread available on the system as well as one configuration for each GPU. Each configuration was tested four times and the mean and standard deviation of the benchmark execution times were recorded. Fig. 4 displays the execution time for each CPU benchmark configuration and a single NVIDIA V100 for comparison.

We determine the theoretical speedup of the processing module using some of the most common mathematical libraries in Python: NumExpr, NumPy, and PyTorch for GPU processing. NumExpr has its own internal scaling mechanism which provides an analog to the PyPWA processing module; for this reason, NumExpr is included twice. NumExpr is included with its own multithreading enabled or disabled (NumExpr ST), both using the Parallel Processing Module. NumExpr ST and multiprocessing with PyPWA provide the best performance overall on CPUs as NumExpr is able to evaluate expressions without returning intermediate results and PyPWA supports data caching. However, using

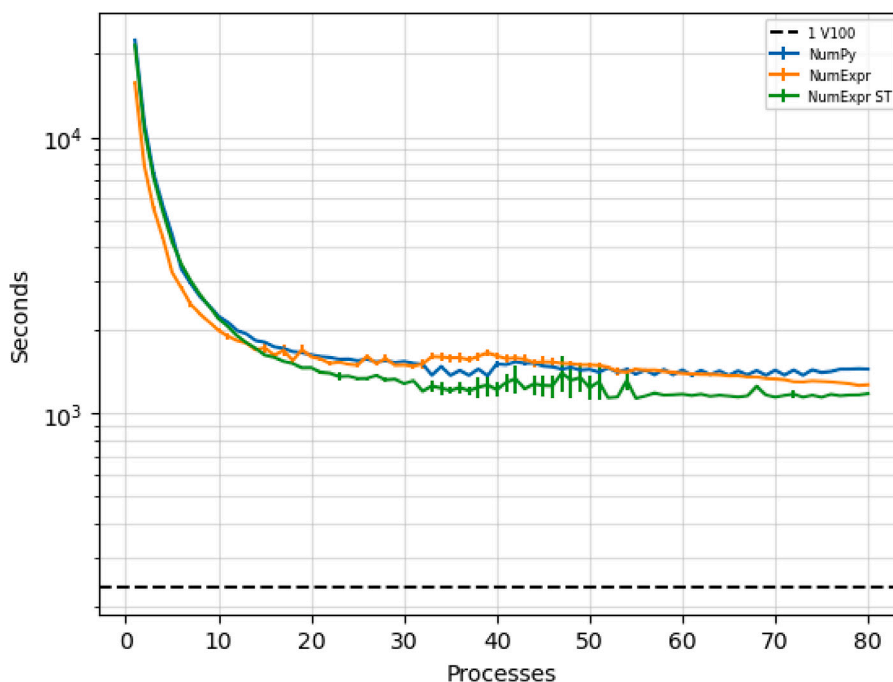

Fig. 4. The total execution time, using different libraries, is shown as a function of the number of processes. For additional comparison, a single V100 GPU data point is shown as a horizontal black dashed line. The benchmark fit was based on Appendix A.

a single V100 GPU performs much better than the fastest 80-thread CPU execution. The Speedup for n threads, $S(n)$, is obtained from:

$$S(n) = \frac{execution\ time\ (n = 1)}{execution\ time\ (n)} \tag{6}$$

The benchmark data are then fitted with the functional form of Amdahl's Law (Eq. (7) [37] to obtain the percentage of the algorithm that is run in parallel (p). The theoretical speedup is given by

$$S(n) = \frac{1}{(1 - p) + (\frac{p}{n})} \tag{7}$$

The closer p is to 1, the better the algorithm scales. Table 2 displays the resulting values for speedup (S) and the percentage of the algorithm that is parallel (p) for each math library for 80 threads on the two CPUs or the three GPUs. Table 3 shows a direct comparison between single-thread execution and execution on the whole CPU using all 80 available threads. With an almost 96% parallel algorithm on CPUs and almost 99% on GPUs, these results show that PyPWA scales well on the benchmark system.

5. Summary

PyPWA provides a flexible set of tools within the Python ecosystem for amplitude analysis of multi-particle final states. The package is built from individual and mostly independent components that the user can arrange in a variety of ways. PyPWA primarily functions as a toolkit and can solve a broad collection of optimization problems. Users provide a function (model), data, and simulation in their preferred data formats. PyPWA provides tools for data processing and analysis. It also provides various ways of speeding up the calculations through parallel processing and user-friendly GPU support with PyTorch.

The flexibility of PyPWA and the use of many standard Python packages make it an ideal tool to perform fits of models to data. The packages are user-friendly to install on Linux and MacOS using Anaconda. The examples provided with the code allow for a quick start and the Python ecosystem comes with a large user base with lots of support. The Python ecosystem is at the root of the PyPWA coding and parallelism is inherent in its design.

CRediT authorship contribution statement

Mark Jones: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Peter Hurck:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **William Phelps:** Writing – review & editing, Writing – original draft, Visualization, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization. **Carlos W. Salgado:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Declaration AI-assisted technologies in the writing process

During the preparation of this work, the authors used ChatGPT in order to do minor copy editing and suggestions for figure captions. After using this service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

Acknowledgments

We would like to acknowledge former members of the PyPWA project who contributed during the past years: Brandon DeMello, Joshua Pond, Christopher Banks, Michael Harris, Stephanie Bramlett,

and Ryan Wright. We would like to thank the Thomas Jefferson National Facility (JLab) computing division for their support, and Prof. G. Hsieh of Norfolk State University (NSU) Computing Sciences Department, for facilitating the use of their HPC cluster. This work was supported in part by the National Science Foundation grants # 0855338, # 1205763 and # 2110797, the Science and Technology Facilities Council (STFC) of the United Kingdom, and the U.S. Department of Energy, Office of Science, Office of Nuclear Physics under contract DE-AC05-06OR23177.

Appendix A. Example: $\eta\pi$ photoproduction

To demonstrate the use of PyPWA, we present a partial wave analysis on simulated data of the reaction $\vec{\gamma} p \to \eta\pi\ p$, the photoproduction of two pseudo-scalar mesons. The goal is to find intermediary meson states (X) that decay via $X \to \eta\pi$. We assume that the reaction $\vec{\gamma} p \to X p$ occurs by diffractive scattering of a linearly polarized photon beam off a proton target which remains intact. All code necessary to run this example is available on our GitHub page [36].

We perform a so-called *mass-independent* partial wave analysis [5], where we include the four-momentum transfer dependency of the production amplitudes in our kinematic simulation. We divide the data and simulation in bins of X meson mass and consider fixed beam energy. Therefore, the two pseudo-scalar intensities, in each mass bin, will have only angular dependencies. We factorize the total transition amplitudes into a production amplitude V (interaction, X production) and a decay amplitude A (decay of X into the final state particles). Production amplitudes are generally unknown at these energies, they are fully (mass-independent) or partially (mass-dependent) fitted to the data. The production amplitudes contain information about the hadronic QCD-based interaction. Those interactions are more difficult to model; therefore, for a mass-independent analysis, the production amplitudes will be considered a *weight* on each decay amplitude. These weights are the parameters to be fitted to the data to obtain the observed overall intensity. Classic partial wave decomposition, truncated to low angular momenta contributions, represents a good first attempt to obtain a set of amplitudes, *the model*. It can then be checked that the data are reasonably described by this model. It can be shown [5], that for two pseudo-scalars, the decay amplitudes are given by spherical harmonics.

We use amplitudes derived by the JPAC [38] collaboration to simulate and consecutively fit the intensities of the given set of resonances and waves. We use a beam with a fixed energy of 8 GeV and 40% linear polarization fraction and an exponential t-Mandelstam distribution $\propto \exp(-bt)$ with $b = 6\,\mathrm{GeV}^{-2}$. The JPAC two pseudo-scalar amplitudes are defined by the following parameters: $\mathscr{P}$ is the polarization fraction, Φ is the polarization angle, and ϵ, L, m are the three quantum numbers of the waves (reflectivity, angular momentum and positive z-component of angular momentum). The two angles θ, ϕ are describing the decay particles in the Helicity frame [38], the total intensity is given by Eq. (A.1) (Eq. (B4) of Ref. [38]).

$$I(\theta,\phi,\mathscr{P},\Phi) = I^{(0)}(\theta,\phi) - \mathscr{P} I^{(1)}(\theta,\phi)\cos 2\Phi$$

$$-\mathscr{P} I^{(2)}(\theta,\phi)\sin 2\Phi \qquad \text{(A.1)}$$

The intensities, $I^{(0)}, I^{(1)}, I^{(2)}$ are calculated from expressions (D12) of Appendix D of Ref. [38].

An example of code used to calculate the JPAC intensities (using PyTorch) is shown below (see full code on GitHub [36]):

```
# Calculate Total Intensity
import numpy as np
import pandas as pd
import scipy.special
import torch as tc
from PyPWA import NestedFunction

# Calculate Decays directly from Spherical Harmonics
def produce_specific_decay(theta, phi, m_waves, l_waves):
    theta[theta < 0] = theta[theta < 0] + 2 * np.pi

    decay = np.empty((len(m_waves), len(theta)), "c16")
    for index, (m, l) in enumerate(zip(m_waves, l_waves)):
        decay[index] = scipy.special.sph_harm(m, l, theta, phi)

    return decay

class FitWithGPU(NestedFunction):

    USE_TORCH = True
    USE_MP = False

    def __init__(self, initial_params):
        super(FitWithGPU, self).__init__()
        self.device = tc.device('cpu')
        self.__alpha = tc.Tensor([])
        self.__pol = tc.Tensor([])
        self.__phi = np.array([])
        self.__theta = np.array([])
        self.__decay = tc.Tensor([])

        elm = make_elm(initial_params)
        self.__e = tc.from_numpy(elm['e'])
        self.__l = elm['l']
        self.__m = elm['m']

    def setup(self, data):
        # Handle Torch Devices based on USE_MP status
        if not self.USE_MP:
            self.device = tc.device("cuda:0")
        self.__alpha = tc.from_numpy(data['alpha']).to(self.device)
        self.__pol = tc.from_numpy(data['pol']).to(self.device)
        self.__e = self.__e.to(self.device)

        self.__decay = tc.from_numpy(produce_specific_decay(
            data['phi'], data['theta'], self.__m, self.__l
        )).to(self.device)
        return self

    def calculate(self, params):
        vs = params[::2] + 1j * params[1::2]
        vs = tc.from_numpy(vs).to(self.device)

        v = vs * self.__decay.T
        v_conj = vs * tc.conj(self.__decay).T

        u10 = v.T[self.__e == 1].sum(0)
        u20 = v_conj.T[self.__e == 1].sum(0)
        u11 = v.T[self.__e == -1].sum(0)
        u21 = v_conj.T[self.__e == -1].sum(0)

        return self.__compute_intensity(u10, u20, u11, u21)

    def __compute_intensity(
            self, u10: tc.Tensor, u20: tc.Tensor,
            u11: tc.Tensor, u21: tc.Tensor) -> tc.Tensor:

        i0 = (
                u10 * tc.conj(u10) + u20 * tc.conj(u20)
                + u11 * tc.conj(u11) + u21 * tc.conj(u21)
        ).real

        i1 = 2 * (-1 * (u10 * tc.conj(u20)).real
            + (u11 * tc.conj(u21)).real)
        i2 = 2 * (-1 * (u10 * tc.conj(u20)).imag
            + (u11 * tc.conj(u21)).imag)

        intensity = i0 - self.__pol * i1 * tc.cos(2 * self.__alpha)
        intensity -= self.__pol * i2 * tc.sin(2 * self.__alpha)

        return intensity.real
```

Table A.4
Resonances/Waves definition for simulation, See text for details.

Mass (GeV/c^2)	Weight	(ϵ, L, M)
0.980	0.5	(1, 0, 0)
1.306	0.3	(1, 2, 1)
1.722	0.2	(1, 1, 1)

We perform the analysis on a simulated sample and assume a detector with full acceptance and 100% efficiency. Using our simulation package, we simulated three resonances decaying to $\eta\pi$, each in a different pure wave, defined by the reflectivity (ϵ), total angular momentum (L) and z-component of the angular momentum (m). The input values are listed in Table A.4.

The simulated X mass distribution, the cosine of the polar helicity angle of the η particle (θ) versus X mass, and the polarization angle versus the azimuthal helicity angle (ϕ) of the η particle are shown in Fig. A.5 . In this example, our analysis goal is to extract the resonances from the simulated data by fitting the $\eta\pi$ angular distributions using the same JPAC amplitudes.

We define the set of waves we will use in the fit and initial values for the production amplitudes. The optimization will be done via an event-by-event extended log-likelihood fit using iMinuit. In this example, we used only positive reflectivity ($\epsilon = 1$), $L < 3$ and $M = 0, 1$ *or* 2, a

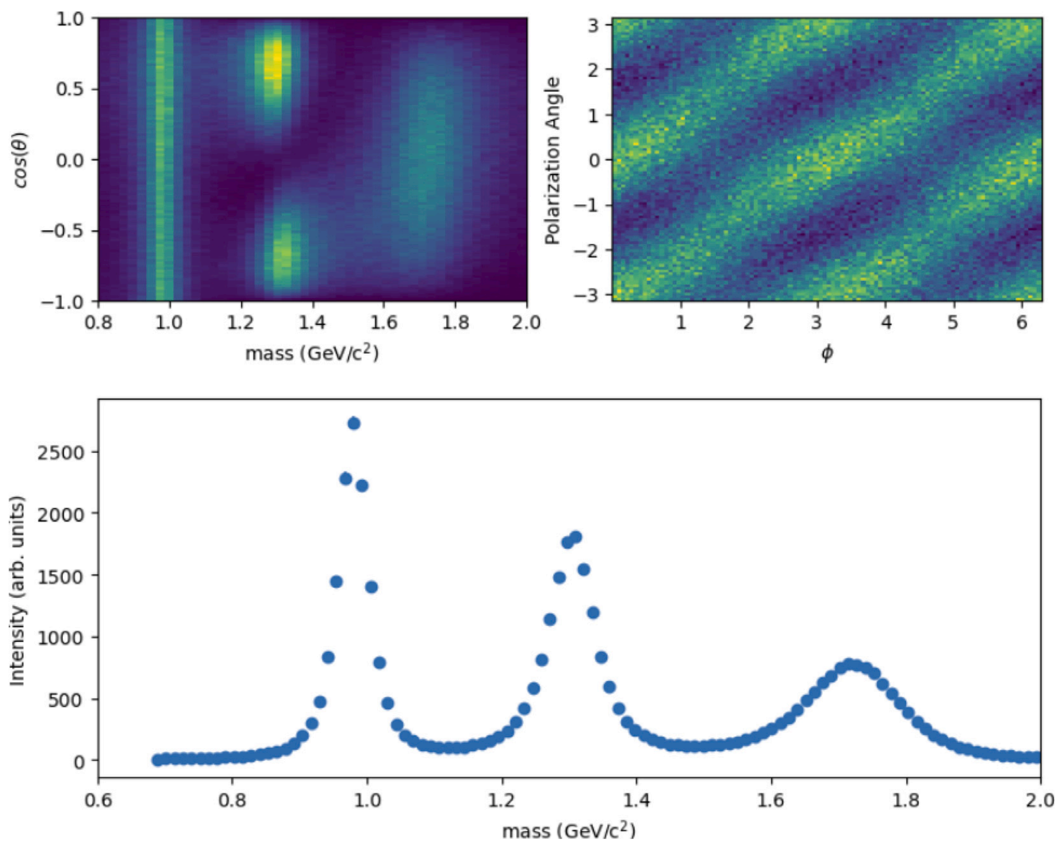

Fig. A.5. Generated mass distribution showing the three input resonances. Also shown are the $\cos(\theta)$ vs. $mass$ and $polarization\ angle$ vs. ϕ for the same generated data. Angles (θ, ϕ) are defined in the Helicity frame.

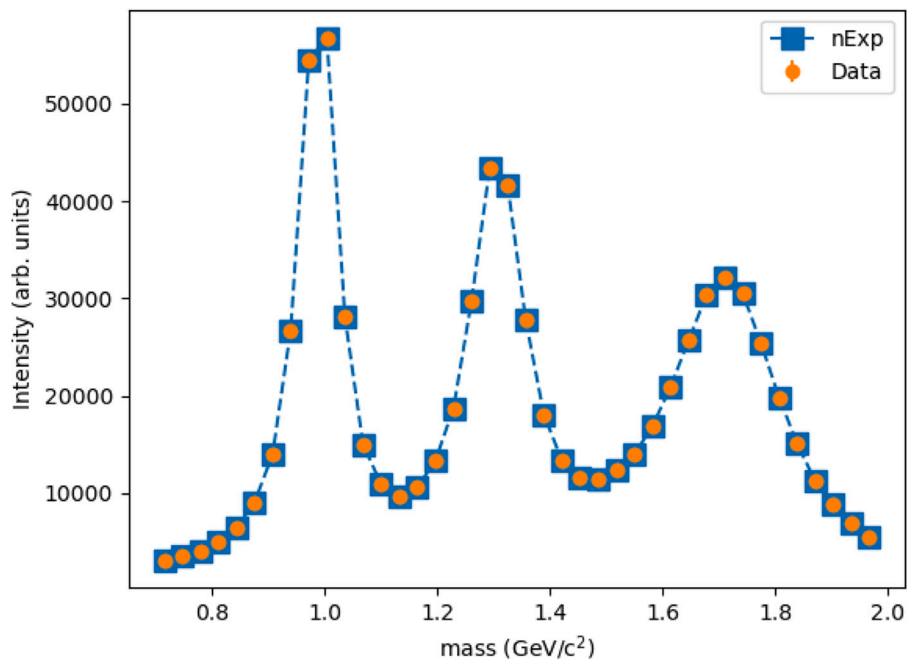

Fig. A.6. The figure shows the fitted total intensity (squares) versus mass. The simulated data points are also plotted (circles). The fitted values and data are in very good agreement.

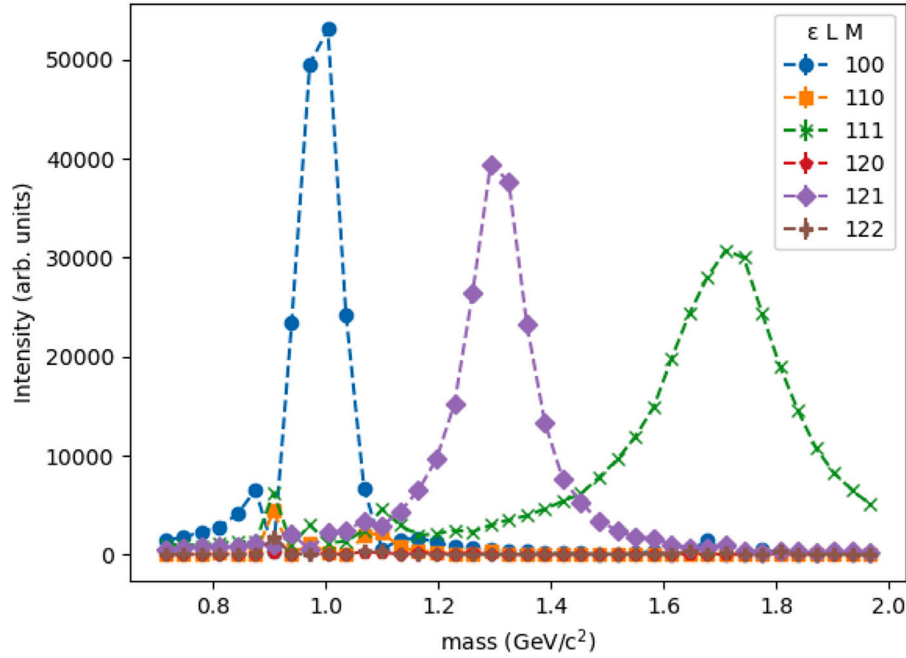

Fig. A.7. The figure shows the fitted intensity versus mass for different waves in the fit set. The fit reproduces the wave composition of the simulated data. Added waves, not present in the simulated data, do not contribute to the intensity.

Table B.5
Specifications of the Dual Cascade Lake Server.

CPU	
Model	Intel Xeon Gold 6230
Architecture	Cascade Lake-SP
Socket	Dual Socket LGA-3647
Cores/Threads	20/40 per CPU
Base frequency	2.1 GHz
Max Turbo frequency	3.9 GHz
Cache	27.5 MB
Memory	
Type	DDR4-2933 ECC
Capacity	512 GB (16 × 32GB)
GPU	
Model	NVIDIA Tesla V100 PCIe
Architecture	Volta
Number of GPUs	3
CUDA cores	5120
Memory	32 GB HBM2

S, P, and D wave set. We also need to use simulated (accepted and generated) samples to obtain the expected and true number of entries in a bin to calculate the extended log-likelihood. The software package provides tools for the user to define the bins, the variable to be binned, the bin ranges, and the extended log-likelihood.

Fitting with iMinuit, we obtain the optimal production amplitudes that minimize the negative extended log-likelihood (c.f. Eq. (5)).

Calculating the expected number of events in a mass bin, we can compare with the simulated data as shown in Fig. A.6. The fitted values and data are in very good agreement. Fig. A.7 shows the intensities versus mass separated in each of the fitted waves, the goal of our analysis. As shown in Fig. A.7, the results of the fit match the input resonances and waves of the simulated data well, i.e, we were able to extract resonances and associated quantum numbers (waves) from the simulated input data.

Appendix B. Benchmark system specifications

Table B.5 shows the specifications of the Dual Cascade Lake Server used for benchmarking the scaling of PyPWA.

References

[1] L. Schiff, Quantum Mechanics, McGraw Hill, 1968.
[2] A. Palano, Light meson spectroscopy and gluonium searches in η_c and $\Upsilon(1S)$ decays at babar, in: 8th Workshop on Theory, Phenomenology and Experiments in Flavour Physics: Neutrinos, Flavor Physics and beyond, 2022, arXiv:2210.00782.
[3] M. Battaglieri, et al., First measurement of direct f0(980) photoproduction on the proton, Phys. Rev. Lett. 102 (2009) 102001, http://dx.doi.org/10.1103/PhysRevLett.102.102001, arXiv:0811.1681.
[4] S. Adhikari, et al., Measurement of spin density matrix elements in $\Lambda(1520)$ photoproduction at 8.2–8.8 GeV, Phys. Rev. C 105 (3) (2022) 035201, http://dx.doi.org/10.1103/PhysRevC.105.035201, arXiv:2107.12314.
[5] C.W. Salgado, D.P. Weygand, On the partial-wave analysis of mesonic resonances decaying to multiparticle final states produced by polarized photons, Phys. Rep. 537 (2014) 1–58, http://dx.doi.org/10.1016/j.physrep.2013.11.005, arXiv:1310.7498.
[6] B. Ketzer, B. Grube, D. Ryabchikov, Light-meson spectroscopy with COMPASS, Prog. Part. Nucl. Phys. 113 (2020) 103755, http://dx.doi.org/10.1016/j.ppnp.2020.103755, arXiv:1909.06366.
[7] 2023, https://www.jlab.org/pypwa.
[8] S. Agostinelli, et al., GEANT4–a simulation toolkit, Nucl. Instrum. Methods A 506 (2003) 250–303, http://dx.doi.org/10.1016/S0168-9002(03)01368-8.
[9] H. Dembinski, P. Ongmongkolkul, C. Deil, H. Schreiner, M. Feickert, C. Burr, J. Watson, F. Rost, A. Pearce, L. Geiger, A. Abdelmotteleb, A. Desai, B.M. Wiedemann, C. Gohlke, J. Sanders, J. Drotleff, J. Eschle, L. Neste, M.E. Gorelli, M. Baak, M. Eliachevitch, O. Zapata, Scikit-hep/iminuit, 2023, http://dx.doi.org/10.5281/zenodo.7750132.
[10] F. James, M. Roos, Minuit: A system for function minimization and analysis of the parameter errors and correlations, Comput. Phys. Comm. 10 (1975) 343–367, http://dx.doi.org/10.1016/0010-4655(75)90039-9.
[11] D. Foreman-Mackey, D.W. Hogg, D. Lang, J. Goodman, Emcee: The MCMC hammer, Publ. Astron. Soc. Pac. 125 (925) (2013) 306–312, http://dx.doi.org/10.1086/670067.
[12] J.P. Cummings, D.P. Weygand, An object-oriented approach to partial wave analysis, 2003, arXiv:physics/0309052v1.
[13] ComPWA, 2023, https://compwa-org.readthedocs.io/.

[14] AmpTools, 2023, https://github.com/mashephe/AmpTools.
[15] GPUPWA, 2016, https://sourceforge.net/projects/gpupwa/.
[16] ROOTPWA, 2020, https://github.com/ROOTPWA-Maintainers/ROOTPWA.
[17] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA, Queue 6 (2) (2008) 40–53, http://dx.doi.org/10.1145/1365490.1365500.
[18] C.R. Harris, K.J. Millman, S.J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith, R. Kern, M. Picus, S. Hoyer, M.H. van Kerkwijk, M. Brett, A. Haldane, J.F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T.E. Oliphant, Array programming with NumPy, Nature 585 (7825) (2020) 357–362, http://dx.doi.org/10.1038/s41586-020-2649-2.
[19] P. Virtanen, R. Gommers, T.E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S.J. van der Walt, M. Brett, J. Wilson, K.J. Millman, N. Mayorov, A.R.J. Nelson, E. Jones, R. Kern, E. Larson, C.J. Carey, İ. Polat, Y. Feng, E.W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E.A. Quintero, C.R. Harris, A.M. Archibald, A.H. Ribeiro, F. Pedregosa, P. van Mulbregt, SciPy 1.0 Contributors, SciPy 1.0: Fundamental algorithms for scientific computing in python, Nature Methods 17 (2020) 261–272, http://dx.doi.org/10.1038/s41592-019-0686-2.
[20] J.D. Hunter, Matplotlib: A 2D graphics environment, Comput. Sci. Eng. 9 (3) (2007) 90–95, http://dx.doi.org/10.1109/MCSE.2007.55.
[21] P. Peterson, F2PY: A tool for connecting fortran and python programs, Int. J. Comput. Sci. Eng. 4 (4) (2009) 296–305, http://dx.doi.org/10.1504/IJCSE.2009.029165.
[22] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, K. Smith, Cython: The best of both worlds, Comput. Sci. Eng. 13 (2) (2011) 31–39, http://dx.doi.org/10.1109/MCSE.2010.118.
[23] F. Pérez, B.E. Granger, IPython: a system for interactive scientific computing, Comput. Sci. Eng. 9 (3) (2007) 21–29, http://dx.doi.org/10.1109/MCSE.2007.53, URL https://ipython.org.
[24] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, J. development team, Jupyter notebooks - a publishing format for reproducible computational workflows, in: F. Loizides, B. Scmidt (Eds.), Positioning and Power in Academic Publishing: Players, Agents and Agendas, IOS Press, 2016, pp. 87–90, URL https://eprints.soton.ac.uk/403913/.
[25] R. Brun, F. Rademakers, ROOT — An object oriented data analysis framework, Nucl. Instrum. Methods Phys. Res. A 389 (1) (1997) 81–86, http://dx.doi.org/10.1016/S0168-9002(97)00048-X, New Computing Techniques in Physics Research V.
[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, PyTorch: An imperative style, high-performance deep learning library, in: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, R. Garnett (Eds.), Advances in Neural Information Processing Systems, Vol. 32, Curran Associates, Inc., 2019, pp. 8024–8035.
[27] J.P. Cummings, D.P. Weygand, An object-oriented approach to partial wave analysis, 2003, arXiv:physics/0309052.
[28] UPROOT, 2020, https://pypi.org/project/uproot/.
[29] S. Gueron, S. Johnson, J. Walker, SHA-512/256 cryptology eprint archive, paper 2010/548, 2010, https://eprint.iacr.org/2010/548.
[30] M. Pivk, F.R. Le Diberder, SPlot: A statistical tool to unfold data distributions, Nucl. Instrum. Methods A 555 (2005) 356–369, http://dx.doi.org/10.1016/j.nima.2005.08.106, arXiv:physics/0402083.
[31] F. James, M. Winkler, MINUIT user's guide, 2004.
[32] D. Foreman-Mackey, Corner.py: Scatterplot matrices in Python, J. Open Source Softw. 1 (2) (2016) 24, http://dx.doi.org/10.21105/joss.00024.
[33] D. Goodger, Docstring processing system framework, 2001, PEP 256 URL https://peps.python.org/pep-0256/.
[34] D. Goodger, Docstring conventions, 2002, PEP 287 URL https://peps.python.org/pep-0287/.
[35] 2023, https://pypwa.readthedocs.io/en/main/.
[36] 2023, https://github.com/JeffersonLab/PyPWA.
[37] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, in: AFIPS '67 (Spring), Association for Computing Machinery, New York, NY, USA, 1967, pp. 483–485, http://dx.doi.org/10.1145/1465482.1465560.
[38] V. Mathieu, M. Albaladejo, C. Fernández-Ramírez, A.W. Jackura, M. Mikhasenko, A. Pilloni, A.P. Szczepaniak, Moments of angular distribution and beam asymmetries in $\eta\pi^0$ photoproduction at GlueX, Phys. Rev. D 100 (2019) 054017, http://dx.doi.org/10.1103/PhysRevD.100.054017.