Master in High Performance Computing

Parallel Markov Chain Generator for GNY and ϕ^4 models: An Implementation aimed for Critical Phenomena Studies.

Supervisors:
Antonello Scardicchio Supervisor,
Ivan Girotto co-supervisor

Candidate:
Jesus Espinoza-Valverde

5th edition
2018–2019

Acknowledgments

I am very grateful with my supervisor Prof. Antonello Scardicchio for his very valueble guidance, and with my co-advisor Ivan Girotto because of his unrelenting atention and help. Finally, I also want to thank Gustavo Ramirez for his advice and his decisive lessons on iterative methods and computational linear algebra.

This is what he'd always known:
The promise of something greater
beyond the water's final horizon.

A.T.

Contents

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations	
GNY	Gross Neveu Yukawa
p.b.c	Periodic Boundary Conditions
a.p.b.c	Antiperiodic Boundary Conditions
DFS	Dynamical Fermion Sampling
CGM	Conjugate Gradient Method
PCGM	Preconditioned Conjugate Gradient Method
FCG	Flexible Conjugate Gradient
MCMC	Markov Chain Monte Carlo
HMC	Hamiltonian Monte Carlo
SLLN	Strong Law of Large Numbers
CLT	Central Limit Theorem
LQCD	Lattice Quantum Chromodynamics
etc.	*etc.*

Chapter 1

Introduction

The ϕ^4 and the Gross-Neveu-Yukawa models have shown to be of great importance in physics. Even though the ϕ^4 model happens to be the most simple interacting quantum field theory, it has proven to be a reference point for the study of spontaneous symmetry breaking and renormalization [21] [23]. Its great importance lies, among other examples, in that it turns out to be intimately related to the description of the Higgs field in the standard model [15], and in the fact that it presents a universality class correspondence with the Ising Model [13]. The Gross-Neveu-Yukawa model is a a quartic interaction fermionic field theory, whose importance resides in the fact that it plays a strong role as toy model for Quantum Chromodynamics [11], it shows dynamical symmetry breaking, it has a spectrum of bound states and it is asymptotically free [7]. The Gross Neveu model with two flavors of Majorana fermions is equivalent to the Thirring model [29].

Lattice simulations of this models have been performed in the past (*e.g.* [13], [24]) using Markov Chain Monte Carlo method (specially making extensive use of the Metropolis-Hasting Algorithm). A further improvement over Metropolis-Hasting called Hamiltonian Monte Carlo has also been put in practice (see e.g. [17]). The HMC algorithm has the feature of combining global moves with high acceptance rates, improving in this way the quality of the produced statistics and the reduction of autocorrelation lengths [8] [22] 4.

Here we developed a software that applies Hamiltonian Monte Carlo and Wilson Fermions (a technique used to dynamically sample the fermion fields [31] [10]) for the generation of Markov Chain series for the ϕ^4 and GNY models in three dimensions and two fermionic flavors, ready for their use in statistical computation of observables and order parameters of these theories for critical phenomena study. Here HPC comes into play because this type of simulations introduces the challenge of dealing with massive linear algebra operations in which the memory and computational bounds are easily reached.

Chapter 2

Theoretical Model Description

The Quantum Field Theory we are interested in is the one described the Gross-Neveu-Yukawa Lagrangian given by equation 2.1, which corresponds to the most general renormalizable version of the the GNY-Lagrangian invariant under $O(N)$ (Global symmetry: $O(N)$ x $\mathbb{Z}_2^{\mathcal{T}}$).

2.1 Model in the Continuum

In the GNY-lagrangian σ corresponds to a scalar field and $\psi_i^{(f)}$ corresponds to N_f two-component *Majorana* fermions; The index f runs throughout the total number N_f of fermionic flavors considered by the model. In this work we are going to restrict N_f (number of fermion flavors) to be an even number in order to avoid the sign problem (see 2.3.1).

$$\mathcal{L}_{\text{GNY}}[\sigma,\psi,\bar{\psi}] = \frac{1}{2}(\partial_\mu\sigma)^2 - \frac{1}{2}m^2\sigma^2 - \frac{\lambda}{4!}\sigma^4 - \frac{1}{2}\sum_{f=1}^{N_f}\bar{\psi}^{(f)}\partial_\mu\gamma^\mu\psi^{(f)} - \frac{1}{2}g\sigma\sum_{f=1}^{N_f}\bar{\psi}^{(f)}\psi^{(f)}. \quad (2.1)$$

Here m, λ and g are coupling constants. Furthermore, notice that the scalar part of this Lagrangian corresponds to the ϕ^4-Lagrangian:

$$\mathcal{L}_{\phi^4}[\sigma,\psi,\bar{\psi}] = \frac{1}{2}(\partial_\mu\sigma)^2 - \frac{1}{2}m^2\sigma^2 - \frac{\lambda}{4!}\sigma^4, \quad (2.2)$$

which is a way more simple model with a pretty well known critical behavior. We consider this expression as a limiting case of the GNY model when $g \to 0$ used form checking purposes.
The corresponding Euclidean action our Lagrangian is given by:

$$\mathcal{S}[\sigma,\psi,\bar{\psi}] = \int d\mathbf{x}\ \ \mathcal{L}[\sigma,\psi,\bar{\psi}], \quad (2.3)$$

where the temporal coordinate is taken as a pure imaginary number (*Wick Rotation*). The partition function Z_{GNY} of then reads

$$Z_{\text{GNY}} = \int \mathcal{D}[\bar{\psi},\psi,\phi]e^{-\mathcal{S}[\bar{\psi},\psi,\phi]}, \quad (2.4)$$

with the **Integration Measure** defined as:

$$\mathcal{D}[\bar{\psi},\psi,\sigma] \stackrel{!}{=} \prod_{\mathbf{x}} d\phi(\mathbf{x}) \prod_{f=1}^{N_f} d\psi^{(f)}(\mathbf{x}) d\bar{\psi}^{(f)}(\mathbf{x}), \quad (2.5)$$

where $\prod_{\mathbf{x}}$ runs over the entire *space-time* $\mathbf{x}$. Finally the vacumm expectation values of an operator O are calculated as:

$$\langle O \rangle = \frac{1}{Z_{\text{GNY}}} \int \mathcal{D}[\bar{\psi}, \psi, \sigma] e^{-S[\bar{\psi}, \psi, \sigma]} O[\bar{\psi}, \psi, \sigma]. \tag{2.6}$$

2.2 Model on the lattice

We move our model to the lattice by systematically discretizing the space–time, the fields and the action. By considering a space-time lattice, a natural cut off for the high frequencies is introduced, which means we end up with a completely finite theory [18].

2.2.1 Discretization of Space-Time and Fields

Our new *space-time lattice* is obtained by replacing the usual Euclidean space-time continuum with a lattice Λ, (i.e., performing $\mathbf{x} \in \mathbb{R}^3 \to \mathbf{n} \in \Lambda$), being Λ given by a set of points such that:

$$\Lambda = \{\mathbf{n} = \ (n_0, n_1, n_2) \ \mid \ n_{0,1,2} = 0, 1, ..., L_{1,2,3} - 1\}, \tag{2.7}$$

with a defining the lattice spacing. After this, our fields are naturally discretized as:

$$\begin{aligned} \sigma(\mathbf{x}) &\to \sigma(\mathbf{n}a) \\ \psi(\mathbf{x}) &\to \psi(\mathbf{n}a) \\ \bar{\psi}(\mathbf{x}) &\to \bar{\psi}(\mathbf{n}a). \end{aligned} \tag{2.8}$$

To work with the lattice boundaries we introduce *periodic-boundary-conditions* (p.b.c) and *antiperiodic-boundary-conditions* (a.p.b.c), for the space-coordinates and the time-coordinate respectively. Mathematically:

$$f(\mathbf{n} + \hat{\mu} L_{\hat{\mu}}) = e^{2\pi i \vartheta_{\hat{\mu}}} f(\mathbf{n}), \tag{2.9}$$

where $\vartheta_0 = 1/2$ for a.p.b.c, and $\vartheta_{1,2} = 0$ for p.b.c. Here $\hat{\mu}$ refers to unit vectors in the main $\hat{0}, \hat{1}, \hat{2}$ directions.

Up to order $\mathcal{O}(a^2)$ we use the following rule to transform our derivatives:

$$\partial_{\hat{\mu}} \psi(\mathbf{x}) \to \frac{\psi(\mathbf{n} + \hat{\mu}) - \psi(\mathbf{n} - \hat{\mu})}{2a}. \tag{2.10}$$

Finally the integrals are treated as sums and the integration measure became well-defined:

$$\int d\mathbf{x} ... \to a^3 \sum_{\mathbf{n} \in \Lambda} ..., \tag{2.11}$$

$$\mathcal{D}[\sigma, \psi, \bar{\psi}] \to \prod_{\mathbf{n} \in \Lambda} d\sigma(\mathbf{n}) \prod_{f}^{N_f} d\psi^{(f)}(\mathbf{n}) d\bar{\psi}^{(f)}(\mathbf{n}). \tag{2.12}$$

2.2.2 Discretization of the Action

Using the scheme of the previous section we write down the discretized action as:

$$\mathcal{S}[\sigma,\psi,\bar{\psi}] = \mathcal{S}_S[\sigma] + \mathcal{S}_F[\sigma,\psi,\bar{\psi}], \tag{2.13}$$

where:

$$\mathcal{S}_S[\sigma] = a^3 \sum_{\mathbf{n}\in\Lambda} \left[-2\kappa \sum_{\hat{\mu}} \sigma_{\mathbf{x}}\sigma_{\mathbf{n}+\hat{\mu}} + \sigma_{\mathbf{n}}^2 + \lambda(\sigma_{\mathbf{n}}^2 - 1)^2 \right], \tag{2.14}$$

$$\mathcal{S}_F[\sigma,\psi,\bar{\psi}] = \sum_{f}^{N_f} \sum_{\mathbf{n},\mathbf{m}\in\Lambda} \sum_{\alpha,\beta} \bar{\psi}^{(f)}(\mathbf{n})_\alpha M(\mathbf{n},\mathbf{m})_{\alpha,\beta} \psi^{(f)}(\mathbf{m})_\beta, \tag{2.15}$$

being the matrix $M(\mathbf{n},\mathbf{m})_{\alpha,\beta}$ is expressed as:

$$M(\mathbf{n},\mathbf{m})_{\alpha,\beta} = -\frac{1}{2} g\sigma(\mathbf{n},\mathbf{m})\, \delta_{\mathbf{n},\mathbf{m}}\delta_{\alpha,\beta} - \frac{1}{2}\sum_{\mu} (\gamma^\mu)_{\alpha,\beta} \frac{\delta_{\mathbf{n}+\hat{\mu},\mathbf{m}} - \delta_{\mathbf{n}-\hat{\mu},\mathbf{m}}}{2a} \tag{2.16}$$

where α and β are spin-indexes. We take $D(\mathbf{n},\mathbf{m})_{\alpha,\beta} = -M(\mathbf{n},\mathbf{m})_{\alpha,\beta}$ and we refer to it as our Dirac operator.

The Doubling Problem

The direct application of the matrix 2.16 induces an issue called *doubling problem*, which arises at the time of computing the fermion propagator in momentum space when we use our previous Dirac operator $D(\mathbf{n},\mathbf{m})_{\alpha,\beta} = -M(\mathbf{n},\mathbf{m})_{\alpha,\beta}$ as starting point. The problem is the appearance of unphysical poles in the propagator related to unwanted extra degrees of freedom [10].
The difficulty is mend by adding an extra term to the momentum-space propagator known as *Wilson term* [31] that removes any extra pole and vanishes in the continuum limit (WilsonTerm $\xrightarrow[a\to 0]{}$ 0). In the space representation the Wilson term takes the form of a new addition to the Dirac operator:

$$D(\mathbf{n}|\mathbf{m})_{\alpha,\beta} = \frac{1}{2}\left[\frac{3}{a} + g\sigma\right] \delta_{\mathbf{n},\mathbf{m}}\delta_{\alpha,\beta} - \frac{1}{4a}\sum_{\mu=\pm 1}^{\pm 3} (\mathbb{I} - \gamma^\mu)_{\alpha,\beta}\delta_{\mathbf{n}+\hat{\mu},\mathbf{m}}, \tag{2.17}$$

where we defined $\gamma_{-\mu} = -\gamma_\mu$. For a full derivation of the previous arguments and equation 2.17 see annexes A.1.

2.2.3 Discretization of the Partition Function

Up to this point, and for a f-number of fermions, we can write the lattice partition function for the GNY theory as:

$$\mathcal{Z}_{\text{GNY}} = \int \prod_{\mathbf{n}\in\Lambda} d\sigma(\mathbf{n})\, e^{-\mathcal{S}_S[\sigma]} \prod_{f}^{N_f} \mathcal{Z}_F^{(f)}[\sigma], \tag{2.18}$$

where the fermionic patition function for the f-flavor is written in the following way:

$$\mathcal{Z}_F^{(f)}[\sigma] = \int d\psi^{(f)}(\mathbf{n}) d\bar{\psi}^{(f)}(\mathbf{n})\ \exp\left(\sum_{\mathbf{n},\mathbf{m}\in\Lambda} \bar{\psi}^{(f)}(\mathbf{n}) D_{\mathbf{n},\mathbf{m}}[\sigma]\psi^{(f)}(\mathbf{m}) \right). \tag{2.19}$$

This integral can be computed analytically as a functional of $D_{\mathbf{n},\mathbf{m}}[\sigma]$ using Grassmann-variables formalism (a result known as Mattews-Salam formula), reading (for a full derivation of the following result see A.2):

$$\mathcal{Z}_F^{(f)}[\sigma] = \det[D(\sigma)]. \tag{2.20}$$

Here we note that each fermion flavor will contribute with a full determinant functional in the total partition function, giving:

$$\mathcal{Z}_{\text{GNY}} = \int \prod_{\mathbf{n}\in\Lambda} d\sigma(\mathbf{n})\, e^{-\mathcal{S}_S[\sigma]}\, (\det[D(\sigma)])^{N_f} \tag{2.21}$$

2.2.4 Discretized Orbservable Integral

With all the machinery developed up to this point we can write an integral expression for the computation of lattice observable mean values $\langle O \rangle$ suitable as starting point for finite computer calculations. From 2.21 we reformulate 2.6 as:

$$\langle O \rangle = \frac{\int \prod_{\mathbf{n}\in\Lambda} d\sigma(\mathbf{n})\, e^{-\mathcal{S}_S[\sigma]}\, (\det[D(\sigma)])^{N_f}\; O[D(\sigma),\sigma]}{\int \prod_{\mathbf{n}\in\Lambda} d\sigma(\mathbf{n})\, e^{-\mathcal{S}_S[\sigma]}\, (\det[D(\sigma)])^{N_f}}. \tag{2.22}$$

Nevertheless, as we will see later, it is possible to achieve further powerful simplifications to effectively tackle with the determinant present in this expression.

2.3 Dynamical Fermion Sampling

As we mentioned before, the integral formula 2.22 serves as a starting point to get an expression suitable for the computation of lattice observables. The peculiarity of this relation resides in the explicit presence of the determinant of the Dirac operator ($\det[D(\sigma)]$), which is a direct symptom of the **non-local** nature of the fermion interactions. This explicit appearance has two main drawback: first, the demanding complexity of implementing a direct determinant solver that does not require the whole Dirac matrix as input (which due to memory limitations is not the most convenient approach), and second, the high computational cost of its actual computation (the determinant has $N!$ contributing terms, which is prohibitively high even for moderate size systems).

This present section concerns with the questions of how to approach this determinant, how to interpret it in a Monte Carlo Sampling Calculation, and how to reformulate it in computationally more convenient way using a dynamical fermion sampling via the introduction of the *pseudofermion* idea.

2.3.1 The Fermion Determinant

During a Monte Carlo sampling approach to compute 2.22 the determinants can be interpreted in two different ways, giving two strikingly different approaches of constructing the sampling algorithm. Nevertheless, as we will see in short, one of them exhibits more convenient and appealing features than the other, specially for reasonable small sampling times.

Determinants as part of the observable

The idea of this first approach is to adsorb de determinant functionals into the observable, in such a way that, in order to compute $\langle O[\sigma,\psi,\bar{\psi}]\rangle$ we sample $(\det[D(\sigma)])^{N_f} O[\sigma,\psi,\bar{\psi}]$ using a probability weight factor of the form $P \propto e^{-\mathcal{S}_S[\sigma]}$.

This procedure, besides being relatively easy to implement, rapidly manifest several inconveniences, the first one is that these determinants naturally exhibit huge variations (typically of several order of magnitudes for moderately large lattices) depending on the scalar field configurations (what actually happens to be a explicit manifestation of the noticeable *non-locality* of this formulation). At the end the sum over all configurations displays large fluctuations around the mean value, which leads to serious numerical instabilities [10]. This suggests that treating the determinant as part of the observable is only justified for low-dimensional small lattices and extremely large statistics.

Determinants as part of the weight factor

The second approach is to compute $\langle O[\sigma,\psi,\bar{\psi}]\rangle$ by sampling $O[\sigma,\psi,\bar{\psi}]$ using a propability weight factor of the form $P \propto e^{-\mathcal{S}_S[\sigma]}(\det[D(\sigma)])^{N_f}$. One potential disadvantage of this approach is that to achieve that probabilistic interpretation, one must ensure that the product of determinants is a real and positive quantity.

We can prove that the realness of $\det[D(\sigma)]$ is indeed ensured by construction: realness of $D(\sigma)$ follows from its γ^5-hermiticity ($D^\dagger = \gamma^5 D \gamma^5$) (for a full proof of this hermiticity feature for our particular Dirac operator see A.3):

$$\det[D]^* = \det[D^\dagger] = \det[\gamma^5 D \gamma^5] = \det[\gamma^5]\det[D]\det[\gamma^5] = \det[D] \tag{2.23}$$

A possible negativity of the determinants product is problematic because it automatically leads to a nonsensical probability distribution. This issue is known as **sign problem**, and is avoided simply by using an even number of fermionic flavors, this guarantees the nonnegativity of the joint distribution function. For example, for two flavors:

$$0 \leq \det[D]\det[D] = \det[D]\det[D^\dagger] = \det[DD^\dagger] \tag{2.24}$$

2.3.2 Pseudofermions

The idea of interpreting the fermion determinants as part of the probabilistic weight factor can be carried out using the convenient and strikingly powerful idea of rewriting the determinants as bosonic gaussian integrals, introducing the concept of *pseudofermion field* [10].
The pseudofermion field concept is based on the fact that both real/complex-variables and grassmann-variables gaussian integrals generate determinants, for instance:

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\cdots\int_{-\infty}^{\infty} dx_1 dx_2 \ldots dx_N \ \mathrm{e}^{-\frac{1}{2}x^T A x + Jx} = \left[\frac{(2\pi)^N}{\det[A]}\right]^{1/2} \mathrm{e}^{\frac{1}{2}J^T A^{-1} J}$$

$$\int\int\cdots\int d\eta_1 d\eta_1' \ldots d\eta_N d\eta_N' \ \mathrm{e}^{\eta' A \eta} = \det[A]$$

This means that for a particular fermion integral one can design a correspondent bosonic integral in complex variables that produce exactly the same result. As an example, we can write for the case of two fermionic flavors:

$$\int \mathcal{D}[\psi,\bar{\psi}]\mathrm{e}^{-\bar{\psi}^{(1)}D\psi^{(1)}-\bar{\psi}^{(2)}D\psi^{(2)}} = (\det[D])^2 = \det[DD^\dagger], \tag{2.25}$$

with a bosonic correspondent of the form:

$$\int \mathcal{D}[\phi_R,\phi_I]\mathrm{e}^{-\phi^\dagger(DD^\dagger)^{-1}\phi} = \frac{\pi^N}{\det[(DD^\dagger)^{-1}]}$$

$$\Rightarrow \quad \pi^{-N}\int \mathcal{D}[\phi_R,\phi_I]\mathrm{e}^{-\phi^\dagger(DD^\dagger)^{-1}\phi} = \det[DD^\dagger]. \tag{2.26}$$

Where we have used $\det[A] = 1/\det[A^{-1}]$. From here we can ensure the relation:

$$\int \mathcal{D}[\psi,\bar{\psi}]\mathrm{e}^{-\bar{\psi}^{(1)}D\psi^{(1)}-\bar{\psi}^{(2)}D\psi^{(2)}} = \pi^{-N}\int \mathcal{D}[\phi_R,\phi_I]\mathrm{e}^{-\phi^\dagger(DD^\dagger)^{-1}\phi} \tag{2.27}$$

So in summary, for two fermion flavors:

$$(\det[D])^2 = \det[DD^\dagger] = \int \mathcal{D}[\psi,\bar{\psi}]\mathrm{e}^{-\bar{\psi}^{(1)}D\psi^{(1)}-\bar{\psi}^{(2)}D\psi^{(2)}} = \pi^{-N}\int \mathcal{D}[\phi_R,\phi_I]\mathrm{e}^{-\phi^\dagger(DD^\dagger)^{-1}\phi} \tag{2.28}$$

In the previous relation $\phi = \phi_R + i\phi_I$ is a N-component complex vector that represents an usual scalar field. This scalar field has the same number of degrees of freedom as the corresponding fermion field and is called *pseudofermion field*.

DFS: Dynamical Fermion Sampling

As we can see from 2.28, the introduction of *pseudofermions* allows to move from the problem of directly computing determinants to the problem of sampling new complex scalar (pseudofermion) fields, this is the key idea of the *Dynamical Fermion Sampling* [10]. Notice that this new sampling will reside in the same grounds than the one of the pure scalar $\sigma[\mathbf{n}]$. Using this, we rewrite 2.22 (for 2 fermion flavors) as:

$$\langle O\rangle = \frac{\int\prod_{\mathbf{n}\in\Lambda} d\sigma(\mathbf{n})d\phi_R(\mathbf{n})d\phi_I(\mathbf{n})\ e^{-\mathcal{S}_S[\sigma]-\phi^\dagger(DD^\dagger)^{-1}\phi}\, O[D(\sigma),\sigma]}{\int\prod_{\mathbf{n}\in\Lambda} d\sigma(\mathbf{n})d\phi_R(\mathbf{n})d\phi_I(\mathbf{n})\ e^{-\mathcal{S}_S[\sigma]-\phi^\dagger(DD^\dagger)^{-1}\phi}} \tag{2.29}$$

Note, however, that *non-locality* of the fermion interaction is still present in this new formulation, this due to the presence of the $(DD^\dagger)^{-1}$ term. This inversion is basically the main source of computational effort for solving 2.29.

Chapter 3

Path Integral Solving: Hamiltonian Monte Carlo

> What works once is a trick; what works twice is a method
>
> *Anonymous*

3.1 Markov Chain Monte Carlo

Markov Chain Monte Carlo is our main workhorse to deal with the actual computation of our path integrals. In this section we delve into the justification for using this technique, and its final implementation in the form of Hamiltonian Monte Carlo.

Why Markov Chain Monte Carlo integration?

We tackle first with the question of why to bother with Monte Carlo (Stochastic) integration, instead of using standard deterministic numerical integration algorithms, such as the Simpson rule or some kind of sophisticated quadrature method. The actual drawback of such kinds of deterministic approaches relies on a phenomenon known as the *curse of dimensionality*, which manifests itself when the basic process of regular subdivision of the integration domain, typical of deterministic methods of numerical integration, is carried out in a space of many dimensions [5]. As illustration, let us suppose we want to integrate the following scalar path integral:

$$\langle O \rangle = \frac{1}{\mathcal{Z}} \int \prod_{\mathbf{n} \in \Lambda} d\sigma[\mathbf{n}] \, \mathrm{e}^{-\mathcal{S}[\sigma]} O[\sigma] \tag{3.1}$$

If we work with a 3 dimensional 3x3x3 lattice then the dimensionality of the integral will be 27. If we subdivide each of these 27 directions in 100 sub-intervals, then we end up with the necessity of computing at least of one value for each of the $100^{27} = 10^{54}$ hypercubes in which the domain is subdivided. It is easy to see that, even with a computer performing at the exa-scale, such task would

take ages of the universe to be done (No need to mention that this explosion with the dimensionality of the number of required operations directly repercutes in the integration error of these deterministic integration methods).

These kind of problems are overcome by the use of stochastic methodologies such as Markov Chain Monte Carlo, which are based on the idea of *importance sampling*: A technique that focuses the sampling mostly at high densities zones of the probability distribution, *that means*, at locations associated with larger (more important) contributions to the overall integral.

3.1.1 MCMC Integration

Roughly speaking, a Markov Chain is defined as a series of states $\{\mathbf{x}^{(0)}\} \to \{\mathbf{x}^{(1)}\} \to \cdots \to \{x^{(N)}\}$ such that the probability of obtaining $\{\mathbf{x}^{(k+1)}\}$ from $\{\mathbf{x}^{(k)}\}$, *i.e.*, the transition probability $q[\{\mathbf{x}^{(k)}\} \to \{\mathbf{x}^{(k+1)}\}]$ is independent from any previous, (or future) state $\{\mathbf{x}^{(\alpha)}\}$. The idea of MCMC is to generate a Markov Chain of states $\{\mathbf{x}^{(k)}\}$ whose histogram of appearances *imitates* the one that would be generated by the probability distribution density $\mathcal{P}[\{\mathbf{x}^{(k)}\}]$ that we are interested to sample, allowing us to tackle calculations that required knowledge of $\mathcal{P}[\{\mathbf{x}^{(k)}\}]$, even though this probability density cannot be written in a precisely analytical way.

Before continuing we have to stress that this *imitation* is successful only if the chain meets the following features:

- **Irreducibility:** Every couple of states are always connected by a finite number of intermediate steps.
- **Aperiodicity:** There is no change of emergence of periodic loops of subsets of states in the chain.
- **Detailed Balance Condition:** The transition probabilities q between any pair of states $\mathbf{x}^{(\alpha)}$ and $\mathbf{x}^{(\beta)}$ should be related by the relation:

$$p[\{x^{(\alpha)}\}]q[\{\mathbf{x}^{(\alpha)}\} \to \{\mathbf{x}^{(\beta)}\}] = p[\{\mathbf{x}^{(\beta)}\}]q[\{\mathbf{x}^{(\beta)}\} \to \{\mathbf{x}^{(\alpha)}\}] \tag{3.2}$$

The central outcome of the whole procedure is that, if a appropriate MCMC sampling algorithm is applied, and a infinite size chain is generated, then the *imitation* becomes exact. With this in mind, a integral of the form

$$I = \int d\mathbf{x}\, \mathcal{P}[\{\mathbf{x}^{(k)}\}]\, O(\mathbf{x}) \tag{3.3}$$

would be *exactly* given by the sum:

$$I = \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} O(\mathbf{x}^{(k)}) \tag{3.4}$$

where N is the chain size and the $\{\mathbf{x}^{(k)}\}$ are generated using MCMC. Even though the exactness of this calculation is ensured by Strong Law of Large Numbers (SLLN) only when $N \to \infty$, we can

approximated the calculation to arbitrary precision using increasingly larger values of N, being the error bounded by the Central Limit Theorem (CLT) [12].

Now we proceed to discuss the standard algorithm used implement MCMC, known as Metropolis-Hastings (Random-Walk Exploration).

3.1.2 Metropolis-Hastings Algorithm (Random-Walk Exploration)

Let us suppose we want to generate a Markov Chain of states $\{\mathbf{x}^{(k)}\}$ that imitates the probability distribution density $\mathcal{P}[\{\mathbf{x}^{(k)}\}]$. If $p(\mathbf{x}^{(k)})$ is a function such that $p(\mathbf{x}^{(k)}) \propto \mathcal{P}[\{\mathbf{x}^{(k)}\}]$ (so we can forget about normalization constants), we can use an auxiliary proposal transition probability $q'(\mathbf{x}^{(k+1)}|\mathbf{x}^{(k)})$ (could be a gaussian for instance), to generate, *on-demand*, the true transition probabilities $q(\mathbf{x}^{(k+1)}|\mathbf{x}^{(k)})$ via the Metropolis-Hastings Algorithm [16]:

1. set $k = 0$
2. sample $\mathbf{x}' \sim \pi^0$, and take $\mathbf{x}^{(k)} = \mathbf{x}'$ (the current state of the chain)
3. choose a $q(\mathbf{x}^{(k+1)}|\mathbf{x}^{(k)})$ such that the Markov chain is irreducible and aperiodic
4. *for* $(n < N, n++)$
   a) generate (propose) $\mathbf{x}' \sim q(\mathbf{x}'|\mathbf{x}_k)$
   b) compute the acceptance probability: $q(\mathbf{x}'|\mathbf{x}^k) = \min\left(1, \frac{p(\mathbf{x}')}{p(\mathbf{x}^{(k)})} \frac{q'(\mathbf{x}^{(k)}|\mathbf{x}')}{q'(\mathbf{x}'|\mathbf{x}^{(k)})}\right)$
   c) generate a uniform between (0,1): U
   d) if $U < \alpha(\mathbf{x}'|\mathbf{x}^{(k)})$ then $\mathbf{x}^{(k+1)} = \mathbf{x}'$, else $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)}$

3.1.3 The need for a more powerful MCMC approach

The **Random-Walk** nickname given to this method comes from the fact that the exploration is driven by a transition probability distribution $q'(\mathbf{x}'|\mathbf{x}^{(k)})$ that has, in practice, nothing to do with the target probability density $\mathcal{P}[\{\mathbf{x}^{(k)}\}]$. This "blindness" of the proposal transition probability respect to geometry of $\mathcal{P}[\{\mathbf{x}^{(k)}\}]$ has the consequence that, for complex-shaped multidimensional distributions, it takes numerous tries for this random-walk exploration to "find" appropriate directions of movement where it can reach (and sample) new high probability states. And even though the algorithm can eventually manage to move along a favorable direction for a few steps, it can easily deviate and move along other no so favorable ones. All these subtle features make the Metropolis-Hasting algorithm "slow" in reaching and finding new far-away and *important* states to sample.

This previous effect is directly reflected in the **autocorrelation length**, that is, the minimum number of ω intermediate states between $\{\mathbf{x}^{(k)}\}$ and $\{\mathbf{x}^{(k+\omega)}\}$ so that they are uncorrelated. In order to produce proper statistics, we should only consider states of the Markov chain that are separated by this correlation length, and this limits the number of suitable configurations from the whole Markov Chain that are appropriate for statistical calculations [12].

In common applications, where the evaluation of $\mathcal{P}[\{\mathbf{x}^{(k)}\}]$ (or a function proportional to it), do not require high computational effort, each step of the random-walk exploration algorithm becomes computationally cheap, and the previous drawback is overcomed simply by enlarging the number of total steps N of the algorithm, up to a point where the obtained statistical quality of the results is satisfactory. Nevertheless, if each evaluation of $\mathcal{P}[\{\mathbf{x}^{(k)}\}]$ is expensive, as it is in our case, for instance, where at each step we must solve the system 4.5 for a extremely large dirac operator $D[\sigma]$, then using a MCMC algorithm able to reduce the correlation length becomes desirable.

In this work we have chosen to replace the usage of the Metropolis-Hastings algorithm by **Hamiltonian Monte Carlo**, which, along with its further refinements is the *De Facto* method for Monte Carlo sampling applied for Lattice Quantum Chromodynamics (LQCD) calculations.

3.1.4 Hamiltonian Monte Carlo (HMC)

The Hamiltonian Monte Carlo algorithm achieves a substantial reduction in the **autocorrelation length** by *inserting knowledge of the geometry* of the probability distribution density that it tries to sample. This knowledge acquisition is possible by the introduction of auxiliary *conjugate momentum variables* for each of the degrees of freedom of the original distribution. The HMC algorithm combines simple serial updates of these artificial momenta parameters with samples of the original distribution variables (*position coordinates*) through the computation of *momentum-position* trajectories following **classical Hamiltonian Dynamics** [5].

In essence, the HMC maps the target distribution density $\mathcal{P}[\mathbf{x}]$ to the phase space $\Gamma[\mathbf{x},\mathbf{p}]$ of a classical dynamics system, and uses the canonical equations of motion

$$\dot{\mathbf{x}} = \frac{\partial \mathcal{H}(\mathbf{x},\mathbf{P})}{\partial \mathbf{P}}, \quad \dot{\mathbf{p}} = -\frac{\partial \mathcal{H}(\mathbf{x},\mathbf{p})}{\partial \mathbf{x}}, \tag{3.5}$$

to reach, by means of orbital movement, new far-away states for $\mathbf{x}$, ensuring larger statistical independency, and by this, reducing the autocorrelation length for the contructed Markov Chain. The momentum variables are sample independently, and rebooted after every new configuration is proposed, allowing a change in the exploring orbit of each step of the algorithm. The artificial Hamiltonian $\mathcal{H}(\mathbf{x},\mathbf{p}) = U(\mathbf{x}) + T(\mathbf{p})$ is introduced by interpreting the probability distribution density $\mathcal{P}[\mathbf{x}]$ as a the *potential energy* $U(\mathbf{x})$, and the *kinetic energy* $T(\mathbf{x})$ is constructed as:

$$T(\mathbf{p}) = \frac{1}{2}\mathbf{p}\cdot\mathbf{p} = \frac{1}{2}\sum_{i=1}^{d} p_i^2, \tag{3.6}$$

where d is the number of degrees of freedom of $\mathcal{P}[\mathbf{x}]$. After that, the joint probability distribution for a extended state $(\mathbf{x},\mathbf{p})$ is then proposed as:

$$P(\mathbf{x},\mathbf{p}) = \frac{e^{-\mathcal{H}(\mathbf{x},\mathbf{p})}}{\mathcal{Z}_{ext}}, \tag{3.7}$$

where $\mathcal{Z}_{ext}$ is the normalization constant (basically the integral of $e^{-\mathcal{H}(\mathbf{x},\mathbf{p})}$ over all the possible domains of $\mathbf{x}$ an $\mathbf{p}$). Finally, in order to meet the detailed balance condition, we use as transition

probability formula:

$$q([\mathbf{x}',\mathbf{p}']\,|\,[\mathbf{x},\mathbf{p}]) = \min\left(1, \exp[-\mathcal{H}(\mathbf{x}',\mathbf{p}') + \mathcal{H}(\mathbf{x},\mathbf{p})]\right) \tag{3.8}$$

To see how the expressions 3.7 and 3.8 fit together the detailed balance condition, we just write it and then replace P and q:

$$\frac{e^{-\mathcal{H}(\mathbf{x},\mathbf{p})}}{\mathcal{Z}_{ext}} \min\left(1, \exp[-\mathcal{H}(\mathbf{x}',\mathbf{p}') + \mathcal{H}(\mathbf{x},\mathbf{p})]\right) = \frac{e^{-\mathcal{H}(\mathbf{x}',\mathbf{p}')}}{\mathcal{Z}_{ext}} \min\left(1, \exp[-\mathcal{H}(\mathbf{x},\mathbf{p}) + \mathcal{H}(\mathbf{x}',\mathbf{p}')]\right), \tag{3.9}$$

which is obviously true.

Properties of HMC

- **Reversibility:** Hamiltonian dynamics ensures that, given a time-evolution operator $\hat{T}_s$ such that $\hat{T}_s[\mathbf{x}(t),\mathbf{p}(t)] = [\mathbf{x}(t+s),\mathbf{p}(t+s)]$, there exists an inverse operator $\hat{T}_{-s}$ such that $\hat{T}_{-s}[\mathbf{x}(t+s),\mathbf{p}(t+s)] = [\mathbf{x}(t),\mathbf{p}(t)]$, always with a *one-to-one* correspondence. This feature is important for sampling purposes because it ensures that the MCMC updates via Hamiltonian Dynamics leaves the target distribution invariant (reversibility of the Markov Chain).

- **Volume Preservation:** An crucial property is that Hamiltonian Dynamics preserves the volume of the phase space (This result is known as *Liouville's Therorem*). This feature is of huge practical importance because it keeps our interpretations of acceptance probabilities always meaningful as time evolve.

3.1.5 Applying HMC to our Path Integrals

In order to apply the Hybrid Monte Carlo Algorithm to our problem we first observe that the following transformation of the partition function:

$$\begin{aligned} \mathcal{Z}_{GNY} &= \int \prod_{\mathbf{n}\in\Lambda} d\sigma(\mathbf{n}) d\phi_R(\mathbf{n}) d\phi_I(\mathbf{n})\ e^{-\mathcal{S}_S[\sigma] - \phi^\dagger (D[\sigma]D^\dagger[\sigma])^{-1}\phi} \\ &\downarrow \\ \mathcal{Z}_{GNY} &= \int \prod_{\mathbf{n}\in\Lambda} dp(\mathbf{n}) \prod_{\mathbf{n}\in\Lambda} d\sigma(\mathbf{n}) d\phi_R(\mathbf{n}) d\phi_I(\mathbf{n}) e^{-\{\frac{1}{2}\sum_{\mathbf{n}\in\Lambda} p_{\mathbf{n}}^2 - \mathcal{S}_S[\sigma] - \phi^\dagger (D[\sigma]D^\dagger[\sigma])^{-1}\phi\}}, \end{aligned} \tag{3.10}$$

leaves any expectation value independent of $p_{\mathbf{x}}$ unchanged. Then we interpret the term in the exponential as a Hamiltonian $H(p,\phi)$:

$$\begin{aligned} H(p,\sigma,\phi = \phi_R + i\phi_I) &= \frac{1}{2}\sum_{\mathbf{n}\in\Lambda} p_{\mathbf{n}}^2 - \mathcal{S}_S[\sigma] - \phi^\dagger (D[\sigma]D^\dagger[\sigma])^{-1}\phi \\ &= H_0(p,\sigma) + V_{\text{ext}}(\phi, D[\sigma]), \end{aligned} \tag{3.11}$$

with $H_0(p,\sigma) = \frac{1}{2}\sum_{\mathbf{n}} p_{\mathbf{n}}^2 + \mathcal{S}_S(\sigma)$ and $V_{\text{ext}}(\phi, D[\sigma]) = \phi^\dagger (DD^\dagger)^{-1}\phi$. Notice that H_0 is the hamiltonian corresponding to the ϕ^4-theory Lagrangian and V_{ext} can be interpreted as an external potential. The next step is to compute the Molecular Dynamics force:

$$\begin{aligned} \mathbf{F} &:= -\nabla_\sigma [H_0(p,\sigma) + V_{\text{ext}}(\phi, D[\sigma])] \\ &= -\nabla_\sigma \mathcal{S}_S[\sigma] - \nabla_\sigma \left(\phi^\dagger (DD^\dagger)^{-1}\phi\right). \end{aligned} \tag{3.12}$$

The pure scalar term can be computed directly from 2.14, and is given by:

$$\mathbf{F}_{\phi^4} = \sum_{\mathbf{n}} \hat{\mathbf{n}} \left[2\kappa \sum_{\mu} \left(\sigma_{\mathbf{n}+\hat{\mu}} + \sigma_{\mathbf{n}-\hat{\mu}} \right) - 2\sigma_{\mathbf{n}} - 4\lambda(\sigma_{\mathbf{n}}^2 - 1)\sigma_{\mathbf{n}} \right], \tag{3.13}$$

The second term can be computed using the following matrix identity $\partial M^{-1}/\partial\omega = -M^{-1}(\partial M/\partial\omega)M^{-1}$:

$$\begin{aligned}
\mathbf{F}_{\text{ext}} &= -\phi^{\dagger}(DD^{\dagger})^{-1}\left(\nabla_{\sigma} DD^{\dagger}\right)(DD^{\dagger})^{-1}\phi \\
&= -\left((DD^{\dagger})^{-1}\phi\right)^{\dagger}\left(\frac{\partial D}{\partial\sigma}D^{\dagger} + D\frac{\partial D^{\dagger}}{\partial\sigma}\right)\left((DD^{\dagger})^{-1}\phi\right) \\
&= -g\sum_{\mathbf{n}}\sum_{\mathbf{m}}\left((D_{\mathbf{n},\mathbf{m}}D^{\dagger}_{\mathbf{m},\mathbf{n}})^{-1}\phi_{\mathbf{n}}\right)^{\dagger}\left(\delta_{\mathbf{n},\mathbf{m}}D^{\dagger}_{\mathbf{m},\mathbf{n}} + D_{\mathbf{n},\mathbf{m}}\delta_{\mathbf{m},\mathbf{n}}\right)\left(((D_{\mathbf{n},\mathbf{m}}D^{\dagger}_{\mathbf{m},\mathbf{n}})^{-1}\phi_{\mathbf{n}}\right)
\end{aligned} \tag{3.14}$$

Finally, the steps of the algorithm are summarized as:

- Sampling the conjugate momenta according to the gaussian probability distribution: $\propto \exp(-p_{\mathbf{x}}^2/2)$.
- Numerical Molecular Dynamics evolution to solve the equations of motion for a *time* interval τ: $(p_0, \sigma_0, \phi_0) \to (p_f, \sigma_f, \phi_f)$.
- Then using Metropolis acceptance step, i.e. accept or reject the new state (p_f, σ_f, ϕ_f) according to the probability:

$$P[(p_0, \sigma_0, \phi_0) \to (p_f, \sigma_f, \phi_f)] = \min[1, \exp(-\Delta H)] \tag{3.15}$$

Chapter 4

Implementation

Here we present the schematic structure of our HMC application for the simulation of the GNY model. For clearness we decided to separate the whole scheme in different levels, being the higher levels contained in the lower ones, in a mamushka-like fashion. At each level we will specify algorithmic details and theoretical subtleties, as well as comments on implementation and benchmarking. The details of the infrastructure used for simulations and development are shown in the Table 4.1.

Architecture:	x86_64
Model name:	Intel(R) Xeon(R) Platinum 8170 CPU @ 2.10GHz
CPU(s):	52
Thread(s) per core:	1
Core(s) per socket:	26
Socket(s):	2
NUMA node(s):	2
CPU GHz:	2.10
L1d cache:	32K
L1i cache:	32K
L2 cache:	1024K
L3 cache:	36608K

Table 4.1: The table contains the specification details of the used infrastructure.

4.1 Level 01: Fields, Matrices and Basic Linear Algebra

This level is related with the implementation of our basic study objects: the fields and the Dirac operator. Besides that, it concerns about their most basic operations: the *vector*, *vector-vector* and *matrix-vector* types. A diagram of this level is shown in Figure 4.1.

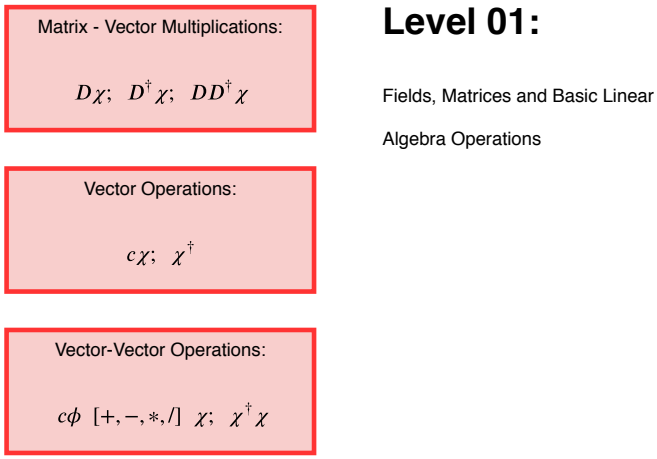

Figure 4.1: Fields, Matrices and Basic Linear Algebra Operations.

4.1.1 Particular subtleties about Fields and Matrices

At this point, where most of our methodology has been established, it is a good time to think about how to deal with our Dirac operator, and to review what are the most basic arithmetic operations that we might apply to it before proposing a strategy to tackle them.

As we will see soon on 4.2, the CGM, PCGM and FCG algorithms (which are going to be used to invert the matrix $DD^\dagger$) and from the force equation 3.14, that the required operations that we need to perform using $D[\sigma]$ are of the type $cDD^\dagger\eta$ and $\eta^\top DD^\dagger\eta$, which can be seen as *matrix-vector* and *vector-vector* if they are computed always from right to left (so avoiding in this way *matrix-matrix* operations). In summary we identify the *matrix-vector* as the main (and only) operation involving $D[\sigma]$ that we will have to care about. To optimize this operation as much as possible (knowing the huge amount of times that it has to be computed for a single simulation), we do need to understand its underlying structure and, in addition, how to take advantage of its sparsity patterns and its symmetry.

Sparsity Profile of $D[\sigma]$

Our Dirac operator $D[\sigma]$ on the lattice is given by:

$$D(\mathbf{n}|\mathbf{m})_{\alpha,\beta} = \frac{1}{2}\left[\frac{3}{a} + g\sigma(\mathbf{n})\right]\delta_{\mathbf{n},\mathbf{m}}\delta_{\alpha,\beta} - \frac{1}{4a}\sum_{\mu=\pm1}^{\pm3}(\mathbb{I}-\gamma^\mu)_{\alpha,\beta}\delta_{\mathbf{n}+\hat{\mu},\mathbf{m}}. \tag{4.1}$$

To visualize the structure of $D[\sigma]$ let's write it in a intuitively simpler way by defining

$$\Gamma_0 := 2^{-1}\left[3a^{-1} + g\sigma[\mathbf{n}]\right]\delta_{\mathbf{n},\mathbf{m}}\delta_{\alpha,\beta} \;\; ; \;\; \Gamma_{\pm\mu} := -(4a)^{-1}(\mathbb{I} - \gamma^{\pm\mu})_{\alpha,\beta}\,\delta_{\mathbf{n}\pm\hat{\mu},\mathbf{m}}, \tag{4.2}$$

so we have:

$$\begin{aligned} D(\mathbf{n}|\mathbf{m})_{\alpha,\beta} = \;& \Gamma_0(\mathbf{n}|\mathbf{m})_{\alpha,\beta} \\ & + \Gamma_{-1}(\mathbf{n}|\mathbf{m})_{\alpha,\beta} + \Gamma_1(\mathbf{n}|\mathbf{m})_{\alpha,\beta} \\ & + \Gamma_{-2}(\mathbf{n}|\mathbf{m})_{\alpha,\beta} + \Gamma_2(\mathbf{n}|\mathbf{m})_{\alpha,\beta} \\ & + \Gamma_{-3}(\mathbf{n}|\mathbf{m})_{\alpha,\beta} + \Gamma_3(\mathbf{n}|\mathbf{m})_{\alpha,\beta}. \end{aligned} \tag{4.3}$$

For the field in a particular lattice position, the diagonal matrix Γ_0 correspond to, let us say, "interactions" of this point with itself, $\Gamma_{\pm1}$ is then related to interactions with lattice points in the $\pm\hat{1}$ directions. The same for $\Gamma_{\pm2}$ in the directions $\pm\hat{2}$ and $\Gamma_{\pm3}$ for the $\pm\hat{3}$ directions. For a space-time lattice of length 3 we display the previous idea in Figure 4.2 (a) and (b) and the sparsity profile of D in (c).

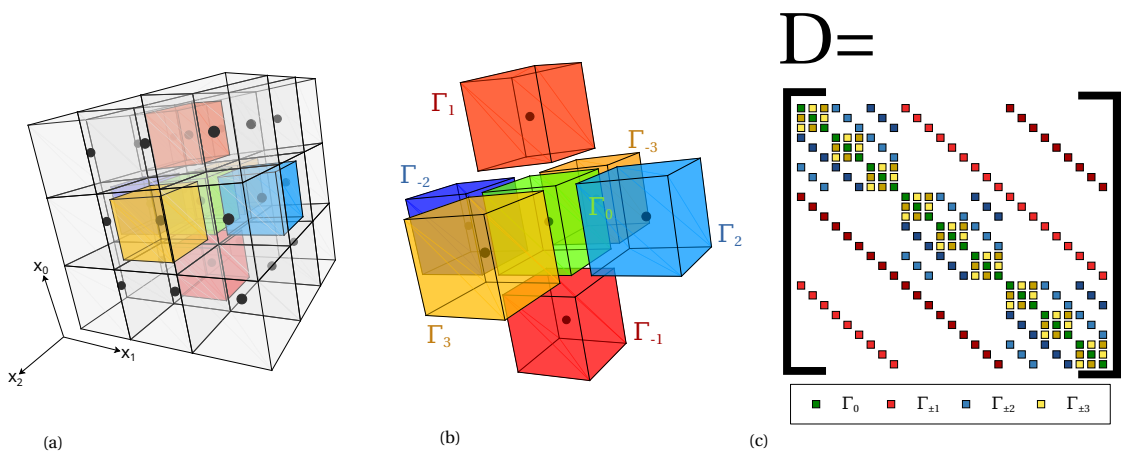

Figure 4.2: (a) 3x3x3 space-time lattice, (b) "interaction" terms for the lattice point $[1,1,1]$ (green) (c) Corresponding matrix profile of D for the whole lattice.

How much memory does $D[\sigma]$ need?

In order to have an idea of how large can become the memory requirement of $D[\sigma]$ for growing lattice sizes, we contemplate 3 scenarios. In the first scenario we consider storing the whole matrix: the number of entries for a lattice size of L is given by $N = 4L^6$, considering that each of this elements is a real complex number, then the memory volume is $64L^6$ bytes. The second case concerns with how much memory is required to store only non-zero terms, that is, considering it as a **sparse matrix**. From 4.3 we see there are 10 non-zero values per row, giving in total $N_{\text{non-zero}} = 20L^3$ non-zero values for the whole matrix (notice how the bigger is the matrix, the sparser it becomes). The memory volume will be given by $320L^3$ bytes. In the third scenario we consider only storing the non-repeated

entries. Considering that the matrices $\Gamma_{\pm 1}$, $\Gamma_{\pm 2}$ and $\Gamma_{\pm 3}$ are repeated from row to row, only Γ_0 contributes appreciably to $N_{non-repeated}$. Finally the we have that $N_{non-repeated} = 18L + L^3$. To write down the required memory volume we have to take into account that the values filling up Γ_0 are given by the scalar field σ, so we can ignore this contribution knowing that they are stored in a different buffer (precisely in a field vector). In this way we obtain a 288 bytes of memory requirement for the non-repeated case, a value that doesn't depend on L. For each of these case we plot the required memory vs the lattice size in Figure 4.3.

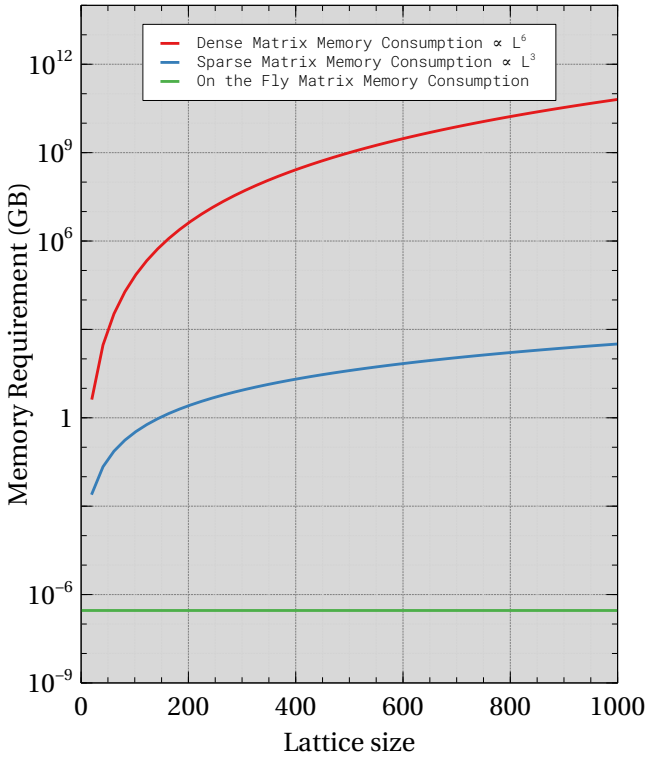

Figure 4.3: Memory requirement in Gigabytes (GB) vs lattice size (L) for the cases [red] Full Maatrix, [blue] Sparse matrix, [green] Non-repeated matrix.

D-vector operations: On-The-Fly approach

Let us consider the operation $D\phi = \phi'$ for a *pseudofermion* field ϕ in the 3x3x3 lattice, and let us focus in how to obtain the middle component $\phi'_{(1,1,1)}$. From Figure 4.4 it is clear that $\phi'_{(1,1,1)} = D[(1,1,1),\cdots]\phi$ can be rewritten as $\phi'_{(1,1,1)} = (D_{diag}[\sigma_{(1,1,1)}] + D_{fix})\eta^{(1,1,1)}\{\phi\}$ by getting rid of all the zeros, so D_{diag}, D_{fix} and $\eta^{(1,1,1)}\{\phi\}$ are much smaller arrays than D and ϕ.

D_{diag} and D_{fix} are of size $7|\beta|^2$ ($|\beta|$ is the spin dimension), D_{diag} depends on σ through Γ_0 and D_{fix} is a totally constant matrix. Note that these features are rather general because they are totally independent of the lattice size. Something similar occurs with $\eta^{\mathbf{n}}\{\phi\}$: it can be constructed taking from ϕ the only necessary entries actually needed for the multiplication. A procedure to construct $\eta^{\mathbf{n}}\{\phi\}$, supposing ϕ is a linear memory buffer with mapping $\phi[n_1,n_2,n_3] \to \phi[L_3L_2\cdot(n_1+\beta_1)+$

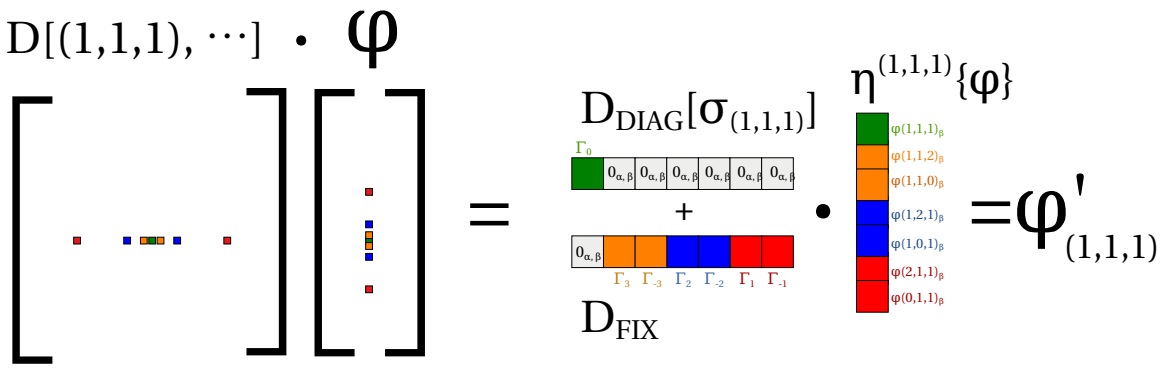

Figure 4.4: Tranformation $\phi'_{(1,1,1)} = D[(1,1,1),\cdots]\phi \rightarrow \phi'_{(1,1,1)} = (D_{diag}[\sigma_{(1,1,1)}] + D_{fix})\eta^{(1,1,1)}\{\phi\}$.

$L_3 \cdot (n_2 + \beta_2) + (n_3 + \beta_3)]$ (β_i are spin indexes) is to move back and forth $\pm|\beta|$, $\pm L_3|\beta|$, $\pm L_3 L_2|\beta|^2$ positions in ϕ from the pointer to $\phi[n_1,n_2,n_3]$, and copy blocks of sizes $|\beta|$ at $\eta, \eta+|\beta|, \eta+2|\beta|, \eta+3|\beta|, \eta+4|\beta|, \eta+5|\beta|, \eta+6|\beta|$ [1]. The total matrix multiplication is the completed by looping over $\mathbf{n} \in \Lambda$, constructing $\eta^{\mathbf{n}}\{\phi\}$ and $D_{diag}[\sigma(\mathbf{n})]$ each time and multiplying $(D_{diag}[\sigma(\mathbf{n})] + D_{fix})\eta^{\mathbf{n}}\{\phi\}$.

The fact that the furtest memory position from $\phi[n_1,n_2,n_3]$ to compute $\phi'[n_1,n_2,n_3]$ is just $L_3 L_2|\beta|^2$ (the total size of the buffer is $L_3 L_2 L_1|\beta|^3$) give us a strong hint on how to parallelize this operations and how to split our domains.

Domain Decomposition and Parallelization

Notice that $L_3 L_2|\beta|^2$ is exactly the size of a 3-2 lattice plane. If we distribute in memory the lattice domain in the $\hat{1}$ direction among the processes, (or in other words, by decomposing along the row-major direction the memory buffers for all our fields and auxiliary vectors), we will see that to compute the multiplication $D \cdot vector$ using the *On-The-Fly* method in a particular subdomain, the elements in the fist (last) lattice planes will need data from the last (first) planes of the previous (next) subdomain. This is a perfect scenario for a *halo exchange* approach: each subdomain can be enlarged with two extra "halo" planes, in such a way that an exchange of planes is carried out before each matrix multiplication, so at every moment each process will have in memory enough data to compute its entire section (see Figure 4.5). MPI non-blocking communication semantics can be used to implement this idea. If each subdomain is large enough then a complete overlap of communication-computation could be achieved.

In the case of *vector-vector* operations this decomposition works fine but with *Reductions* and *AllReductions* instead of *halo exchanges*.

[1] Some modulo operations are needed to avoid going out of the ϕ buffer limits. In this same regard, the negative signs introduced in the entries of D by the aperiodic boundary conditions can be moved from D to η for the convenience of keeping D_{fix} constant.

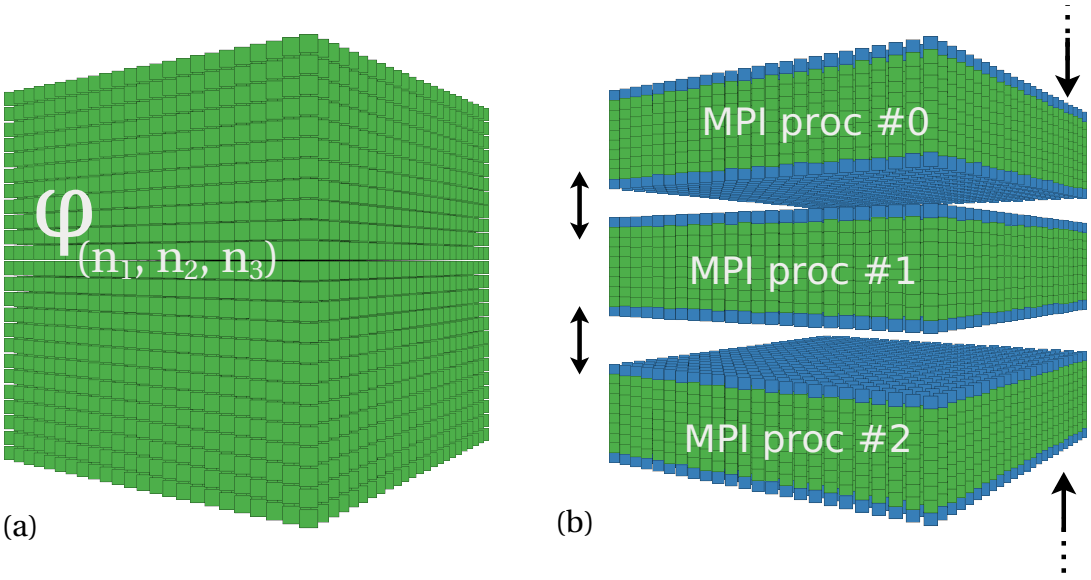

Figure 4.5: (a) Example of an entire lattice domain, (b) Domain descomposition (green), with halos (blue), the double head arrows represents the halo exchange among the processes.

4.1.2 Implementation of Fields

We implemented the fields as c++ classes templated in the type of variable stored (real or complex) and in the number of spinorial degrees of freedom (0,1,...). For instance, if we need to fill up a 3-dimensional space-time lattice of size L with two scalar field σ and three pseudofermion fields ϕ_A and ϕ_B, we declare them as:

```
// Scalar Field: 1-spin-component and real values.
Field<1, double>          sigma_A{L}, sigma_B{L};
// Pseudofermion Field: 2-spin-components and complex values.
Field<2, dcmplx>          phi_A{L}, phi_B{L}, phi_C{L};
```

Listing 4.1: Declaration of Fields.

Now, in order to initialize them we have 3 options: Gaussian random initialization (mean value 0.0, standard deviation 1.0) or uniform initialization (all values equal to 1.0), and finally copy assignment:

```
// Gaussian Initialization of the Fields
gaussianRandomInit(sigma_A, random_seed_A);
gaussianRandomInit(phi_A, random_seed_A + 1);
// Uniform Initialization of Fields.
gaussianRandomInit(sigma_B, random_seed_B);
gaussianRandomInit(phi_B, random_seed_B + 1);
// Copy Assigment
phi_C = phi_B;
```

Listing 4.2: Initialization of Fields.

As *vector* operations we implemented multiplication by a constant and complex conjugation:

```
// Multiplication by a contant (real or complex)
sigma_A *= r ;
phi_A *= c ;
// Complex Conjugation
phi_B.conjugate();
```

Listing 4.3: Vector operations.

For the *vector-vector* operations we implemented summation, subtraction, element-wise multiplication and division, comparison and scalar product.:

```
phi_A = phi_C
// [+,-,/,*, ==] operations
phi_A += phi_B;
phi_A -= phi_B;
phi_A *= phi_B;
phi_A /= phi_B;
bool comp = (phi_A == phi_C);
// Scalar product
dcmplx = phi_B * phi_C;
```

Listing 4.4: Vector-Vector operations.

Finally include a testing module to check for the correctness of these operations:

```
static char *all_tests()
{
     my_run_test(test_populate_scalar);
     my_run_test(test_populate_pseudofermion);
     my_run_test(test_copy_scalar);
     my_run_test(test_copy_pseudofermion);
     my_run_test(test_sum_eq_scalar);
     my_run_test(test_sum_eq_pseudofermion);
     my_run_test(test_min_eq_scalar);
     my_run_test(test_min_eq_pseudofermion);
     my_run_test(test_prod_eq_scalar);
     my_run_test(test_prod_eq_pseudofermion);
     my_run_test(test_dot_prod_pseudofermion);
     my_run_test(test_comparison_scalar);
     my_run_test(test_comparison_pseudofermion);
     return 0;
}
```

Listing 4.5: Fields testing.

Distributed-Memory Fields

A distributed-memory (MPI) version of the field class was also implemented using the description suggested for *domain decomposition* given in 4.1.1. The interface of usage is essentially the same as

the one for *fields*, with the exception that it requires specifying the MPI communicator as argument.

```
// MPI Initialization
MPICommManager                          world(&argc, &argv, thisIsWorld);
// Distributed Scalar Field: 1-spin-component and real values.
Distributed_Field<1, double>            sigma{L, &world};
// Distributed Pseudofermion Field: 2-spin-components and complex values.
Distributed_Field<2, dcmplx>            phi_{L, &world};
```

Listing 4.6: Declaration of Distributed Fields.

Benchmarking

To compare the performance of the multi-threaded and MPI versions of the fields, we show below, in the Figure 4.6, the execution times of a typical *vector-vector* operation (specifically the element-wise multiplication $v_1^{(i)} \leftarrow v_1^{(i)} * v_2^{(i)}$, which appears frequently in the Conjugate Gradient Algorithm), for an increasing number of mpi-processes/threads and for field sizes given by $L = 32, 64, 80$ (vector size given by $N = 4L^6$). As we can see, in all cases the two forms of parallels show comparable execution and scaling times, the MPI version being slightly better.

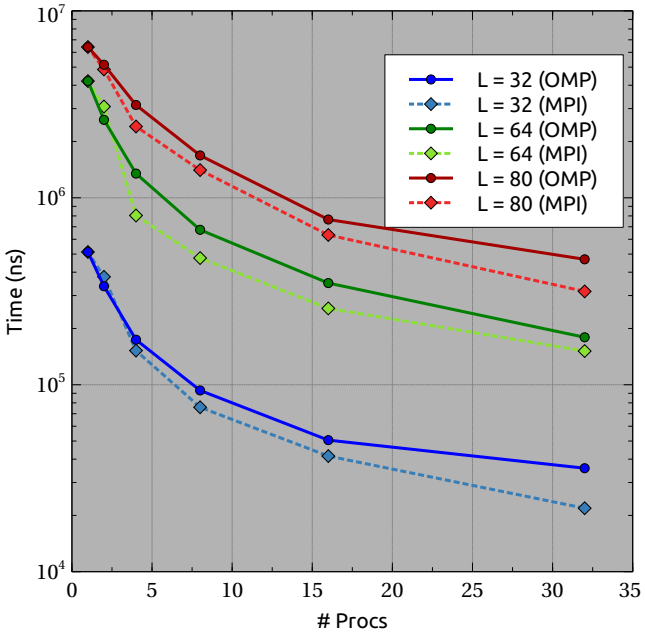

Figure 4.6: Execution times of vector-vector operation $v_1^{(i)} \leftarrow v_1^{(i)} * v_2^{(i)}$ for field sizes $L = 32, 64, 80$ and for $4, 8, 16, 32$ mpi-processes/threads.

4.1.3 Implementation of Dirac Operator

We implemented the operator $D[\sigma]$ and the operations $D\phi$, $D^\dagger\phi$ and $DD^\dagger\phi$ using three different main approaches:

1. Full matrix construction, multiplication using the highly optimized **intel mkl cBLAS** library..

2. Sparse matrix construction (storing only non zero terms), multiplication using the highly optimized **intel mkl Sparse cBLAS** library.

3. Storing only non-repeated terms in a reduced matrix ($D_{diag} + D_{fix}$), generation of profile matrix during multiplication [proposed On-The-Fly method](see 4.1.1).

Given the possibility that each of these approaches had advantages over the others in different lattice size ranges, we implemented all of them so that they had the exact same usage interface. This would allow us to switch from one method to another depending on the range, without implying an alteration of the general code of simulation. This behavior is achieved using c++ object-oriented programming and polymorphism:

```
// Scalar Field: 1-component and real values.
Field<1, double>          sigma{L};
// Pseudofermion Field: 2-component and complex values.
Field<2, dcmplx>          phi{L};
// Output Pseudofermion fields (\phi_out = DD^\dagger \phi)
Field<2, dcmplx>          phi_out1{L}, phi_out2{L}, phi_out3{L};

// Gaussian Initialization of the Fields
gaussianRandomInit(sigma, random_seed_A);
gaussianRandomInit(phi, random_seed_B);

// Creation of the different matrix schemes (Dense, sparse,
// reduced [On-The-Fly])
DenseDiracMatrix            D_dense{L, sdim, gcc, sigma};
SparseDiracMatrix           D_sparse{L, sdim, gcc, sigma};
ReducedDiracMatrix          D_reduced{L, sdim, gcc, sigma};

// Multiplication interface
mult_DDdgg_v(D_dense, phi, phi_out1);
mult_DDdgg_v(D_sparse, phi, phi_out2);
mult_DDdgg_v(D_reduced, phi, phi_out3);

// Overloading of operator== to check if outputs are the same
bool check_against_BLAS          = (phi_out1 == phi_out3);
bool check_against_sparse_BLAS   = (phi_out2 == phi_out3);

// Print check
std::cout << check_with_BLAS  << "\n";
std::cout << check_with_sparse_BLAS  << "\n";
```

Listing 4.7: Basic interface for the three multiplication methods.

Finally include a testing module to check for the correctness of these operations:

```
static char *all_tests()
{
  my_run_test(test_populate_full_D);
```

```
    my_run_test(test_populate_sparse_D);
    my_run_test(test_populate_Reduced);
    my_run_test(test_D_v_full_D);
    my_run_test(test_D_v_sparse_D);
    my_run_test(test_D_v_Reduced_D);
    my_run_test(test_Ddgg_v_full_D);
    my_run_test(test_Ddgg_v_sparse_D);
    my_run_test(test_Ddgg_v_Reduced_D);
    my_run_test(test_DDdgg_v_full_D);
    my_run_test(test_DDdgg_v_sparse_D);
    my_run_test(test_DDdgg_v_Reduced_D);
    return 0;
}
```

Listing 4.8: Dirac Operators testing.

Distributed-Memory Dirac Operator

A distributed-memory (MPI) version for the Dirac Operator was implemented for the *On-The-Fly* (Reduced) approach, following the description given in 4.1.1, and using the just mentioned distributed-memory fields,

```
// MPI Initialization
MPICommManager                  world(&argc, &argv, thisIsWorld);
ConditionalOStream              pcout{std::cout, rank == 0};

// Distributed Fields declaration
Parallel_Field<1, double>       pscalar1{L, &world};
Parallel_Field<2, dcmplx>       ppseudofermion1{L, &world};
Parallel_Field<2, dcmplx>       ppseudofermion2{L, &world};

// Distributed Fields Initialization
pscalar1.gaussianRandomInit(random_seed_A);
ppseudofermion1.gaussianRandomInit(random_seed_B);

// Distributed On-The-Fly [reduced] matrix Declararion-Initialization
ParallelReducedDiracMatrix      pD_reduced{L, sdim, gcc, pscalar1, &world};

// Parallel (MPI) Multiplication.
pD_reduced.mult_D_v(ppseudofermion1, ppseudofermion2);
```

Listing 4.9: Basic interface for distributed memory On-The-Fly Dirac Operator.

Benchmarking

To compare the performance of the multi-threaded (OMP on-the-fly multiplication and MKL Sparse cBlas) and MPI versions of the *matrix-vector* operation, we show below, in the Figure 4.7, the execution

times to perform $DD^{\dagger}\phi$ for an increasing number of MPI-processes/threads and for field sizes given by $L = 32, 64$ (vector size given by $N = 4L^6$). As we can see, in spite of showing a poorer scaling compared to the homemade OMP and MPI versions of the On-the-fly multiplication, the MKL Sparse cBlas implementation shows a better performance, being approximately an order of magnitude faster when only one or two processes / threads are used.

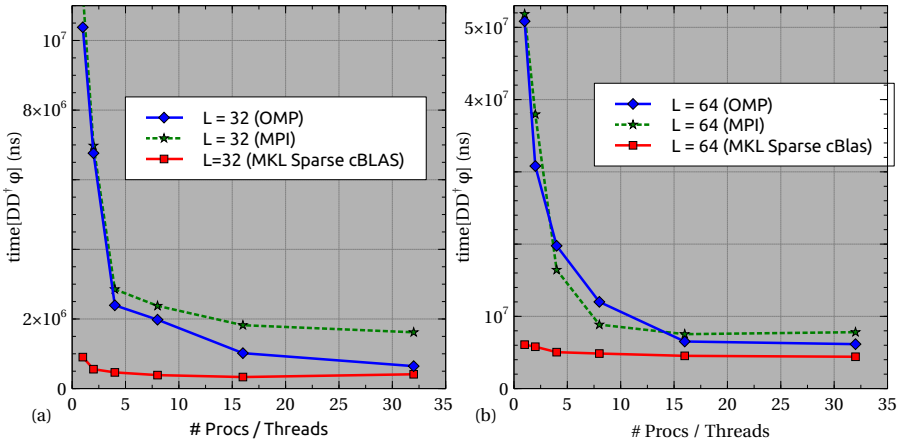

Figure 4.7: Execution times of *matrix-vector* operation $DD^{\dagger}\phi$ for field sizes $L = 32, 64$ and for $4, 8, 16, 32$ mpi-processes/threads.

An interesting behavior occurs in the case of $L = 32$ when the number of MPI-processes/threads equals the size of the field edge (procs = threads = 32) [see Figure (a)]: for the MPI implementation, a poorer scaling compared to the OMP case is observed, which makes sense noticing that when the number of processes approach 32 the size of the *exchange halos* tends to be similar (or equal) to that of the domain subregion. In this scenario the scaling is limited by the redundant workload associated to compute the multiplication sections correspondent to the halos. This problem limits the maximum number of processes that can be used for smaller lattices and only ceases to be serious when the lattice size is several times bigger than the maximum number of processes used [see for instance Figure (b) where the differences between OMP and MPI approaches become less dramatic]. A solution to this problem can be found in the implementation of a more sophisticated domain decomposition procedure, for example using of **MPI Virtual Topologies**.

The redundant workload problem does not affect the OMP implementation, and we consider it more preferable for small or medium lattice sizes over the MPI implementation. The MKL Sparse solution is, however, preferable over these last two, given the fact that it considerably exceeds the performance of the former ones, especially for few threads/processes. This makes it the most appropriate option if we want to run series of independent simulations, each with one or few threads, during a sweeping of couplings parameter space.

In a scenario where we have very large lattices that exceed the RAM capacity of the infrastructure (and therefore also the size of the Sparse matrix), then an MPI implementation (or hybridized with OMP) would become the option to follow.

In this project we are mainly interested in small and medium lattices so an MPI implementation that covers the rest of the levels of our code will be left as future work.

4.2 Level 02: Solving $\phi^\dagger(DD^\dagger)^{-1}\phi$: Conjugate Gradient Methods and Preconditioning

In order to implement the *Dynamical-Fermion-Sampling* approach (DFS) described in 2.3.2 we notice that:

$$\phi^\dagger(DD^\dagger)^{-1}\phi = [\phi^\dagger(D^\dagger)^{-1}][D^{-1}\phi] = [D^{-1}\phi]^\dagger[D^{-1}\phi], \tag{4.4}$$

then, if we define $\chi = D^{-1}\phi$ and sample it according to $\exp(-\chi^\dagger\chi)$, we can recover ϕ from:

$$\phi = D\chi. \tag{4.5}$$

Roughly speaking, we can say that, with DFS we translate the problem of computing the determinant of the Dirac operator (and all the numerical instabilities of doing so) into the problem of sampling new ϕ_I and ϕ_R fields solving the linear system 4.5.

To solve 4.5 we have chosen the Conjugate Gradients method (CGM), which is a iterative method known for having smaller complexity than other solvers such as Gauss or Gauss-Seide, besides being really convenient because its core is based only on simple *vector-vector* and *matrix-vector* operations. By construction he have that $DD^\dagger$ is a symmetric positive-definite (SPD) hermitian matrix, which is the necessary conditions for CGM to work.

4.2.1 Conjugate Gradient Method

The conjugate Gradient Algorithms is given by 28:

$$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0 \tag{4.6}$$

if $\mathbf{r}_0$ is sufficiently small, then return $\mathbf{x}_0$ as the result (4.7)

$$\mathbf{p}_0 := \mathbf{r}_0 \tag{4.8}$$

$$k := 0 \tag{4.9}$$

repeat (4.10)

$$\alpha_k := \frac{\mathbf{r}_k^\top \mathbf{r}_k}{\mathbf{p}_k^\top \mathbf{A}\mathbf{p}_k} \tag{4.11}$$

$$\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{4.12}$$

$$\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k \tag{4.13}$$

(4.14)

if $\mathbf{r}_{k+1}$ is sufficiently small, then exit loop (4.15)

$$\beta_k := \frac{\mathbf{r}_{k+1}^\mathsf{T}\mathbf{r}_{k+1}}{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k} \tag{4.16}$$

$$\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k \tag{4.17}$$

$$k := k+1 \tag{4.18}$$

end repeat (4.19)

return $\mathbf{x}_{k+1}$ as the result (4.20)

4.2.2 Preconditioned Conjugate Gradient

The preconditioning is a method focused to improve the conditiong number of a matrix. The basic principle is to transform the problem $A\mathbf{x} = \mathbf{b}$ into $M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}$ in such a way that $\kappa(M^{-1}A) \ll \kappa(A)$ [3]. A good preconditioning matrix M should approximate A as much as possible be way more easy to invert. In this way $M^{-1}A\mathbf{x} \approx A^{-1}A\mathbf{x} \approx \mathbf{x}$ and $M^{-1}\mathbf{b} \approx A^{-1}\mathbf{b}$ and the new system imitates the trivial problem $\mathbf{x} = A^{-1}\mathbf{b}$ [25].

The issue with using this idea directly with the Conjugate Gradient method is that, although we can ensure M and A to be hermitian by their own, the product $M^{-1}A$ in general is not. This problem could be overcome writing M as $M = EE^T$, then:

$$E^{-T}(E^{-1}AE^{-T})(E^T\mathbf{x}) = E^{-T}E^{-1}\mathbf{b} \tag{4.21}$$

Introducing $\hat{\mathbf{x}} = E^T\mathbf{x}$ we have $E^{-1}AE^{-T}\hat{\mathbf{x}} = E^{-1}\mathbf{b}$. This is a suitable problem because the transformation $E^{-1}AE^{-T}$ preserves the hermiticity of A. Now the problem is to compute E. This could be avoided doing the replacements $r_i \leftarrow E^{-1}r_i$ and $p_i \leftarrow E^T p_i$ in the original Conjugate Gradient algorithm. We end up with [28]:

Problem: solve $\mathbf{Ax} = \mathbf{b}$ with $\mathbf{A}$ Hermitian and positive-definite.

$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$ (4.22)

if $\mathbf{r}_0$ is sufficiently small, then return $\mathbf{x}_0$ as the result

$\mathbf{z}_0 := \mathbf{M}^{-1}\mathbf{r}_0$

$\mathbf{p}_0 := \mathbf{r}_0$ (4.23)

$k := 0$ (4.24)

(4.25)

repeat

$$\alpha_k := \frac{\mathbf{r}_k^\top \mathbf{z}_k}{\mathbf{p}_k^\top \mathbf{A}\mathbf{p}_k} \tag{4.26}$$

$$\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{4.27}$$

$$\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k \tag{4.28}$$

if $\mathbf{r}_{k+1}$ is sufficiently small, then exit loop

$$\mathbf{z}_{k+1} := \mathbf{M}^{-1}\mathbf{r}_{\mathbf{k+1}}$$

$$\beta_k := \frac{\mathbf{z}_{k+1}^\top \mathbf{r}_{k+1}}{\mathbf{r}_k^\top \mathbf{r}_k} \tag{4.29}$$

$$\mathbf{p}_{k+1} := \mathbf{z}_{k+1} + \beta_k \mathbf{p}_k \tag{4.30}$$

$$k := k + 1 \tag{4.31}$$

end repeat

return $\mathbf{x}_{k+1}$ as the result

Using this methodology we implemented three preconditioners, an identity (or dummy) preconditioner used for comparison purposes, and inverse diagonal preconditioner and Hermitian Successive Over-Relaxation.

4.2.3 Preconditioning that changes in every iteration: Flexible Conjugate Gradient

If we want to use a preconditioner that changes according to the results of every interation (so $M = M(\mathbf{r}_i)$), we only need to change the formula for β_k in the original Preconditioned Conjugate Gradient with the Polak-Ribiere formula:

$$\beta_k := \frac{\mathbf{z}_{k+1}^T(\mathbf{r}_{k+1} - \mathbf{r}_k)}{\mathbf{z}_k^T \mathbf{r}_k} \tag{4.32}$$

This allow us to use, for instance, **Conjugate Gradient** and the **Jacobi method** as preconditioners [25].

4.2.4 Hermitian Successive Over-Relaxation

This method requires the decomposition of our Hermitian Matrix A into the form $A = D + L + L^\dagger$ where the matrix D corresponds to the Diagonal and L to the lower triangular. The preconditioning matrix is given by [25]:

$$M(\omega) = \frac{\omega}{2-\omega}\left[\frac{1}{\omega}D + L\right] D^{-1} \left[\frac{1}{\omega}D + L\right]^\dagger \tag{4.33}$$

where ω is an adjustable parameter. Then the preconditioned matrix $M^{-1}A$ takes the form:

$$M^{-1}(\omega)A = \frac{2-\omega}{\omega}\left[\frac{1}{\omega}D + L\right]^{-\dagger} D \left[\frac{1}{\omega}D + L\right]^{-1} A \tag{4.34}$$

The inverse matrix operations $\left[\omega^{-1}D+L\right]^{-1}\mathbf{x}$ and $\left[\omega^{-1}D+L\right]^{-\dagger}\mathbf{x}$ could be computed without the need of a solver; they are obtained using back and forward subtitution. In the case of $\left[\omega^{-1}D+L\right]^{-\dagger}\mathbf{x}$ the system takes the form:

$$\begin{aligned} a_{11}^{(0)}x_1 + a_{12}^{(0)}x_2 + a_{13}^{(0)}x_3 + \cdots + a_{1n}^{(0)}x_n &= b_1^{(0)} \\ a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \cdots + a_{2n}^{(1)}x_n &= b_1^{(0)} \\ \ddots \qquad\qquad = &\ \vdots \\ a_{nn}^{(n-1)}x_n &= b_n^{(n-1)} \end{aligned} \tag{4.35}$$

and the formulas to solve this (backward substitution) become:

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}; \qquad x_i = \frac{b^{(i-1)} - \sum_{j=i+1}^{n} a_{ij}^{(i-1)}x_j}{a_{ii}^{(i-1)}}; \qquad i = n-1, n-2, \ldots, 1 \tag{4.36}$$

The way to solve $\left[\omega^{-1}D+L\right]^{-1}\mathbf{x}$ (forward subtitution) is complete analogous but starting from the upper-left corner instead of starting in the lower-right one.

In Figure 4.8 we plot the condition number $\kappa_2[M^{-1}(\omega)DD^\dagger]$ against ω for the $DD^\dagger$ matrix corresponding to a lattice size of 6 and a coupling constant $g = 1.0$. The condition numbers $\kappa_2[DD^\dagger])$ (original matrix) and $\kappa_2[(\text{Diag})^{-1}DD^\dagger]$ are plotted as horizontal lines for comparison purposes. These calculations were done using Python.

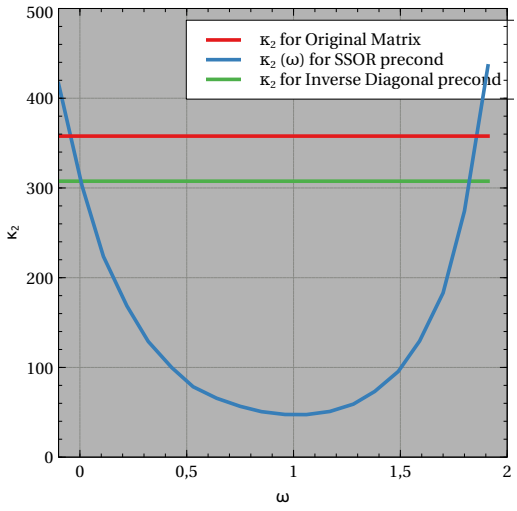

Figure 4.8: HSOR condition number $\kappa_2[M^{-1}(\omega)DD^\dagger]$ plotted against ω for a Lattice size of $L = 6$ and coupling constant $g = 1.0$, $\kappa_2[DD^\dagger])$ and $\kappa_2[(\text{Diag})^{-1}DD^\dagger])$ are also shown.

From Figure 4.8 we see that a $\omega \approx 1.0$ reduces the condition number considerably with respect the original condition number and the one obtained using the inverse diagonal preconditioning.

4.2.5 Implementation of Solvers

As mentioned in 4.2 we proposed solving inverse operation $[DD^{\dagger}(\sigma)]^{-1}$ by means of CGM, PCGM, and FCG (see Figure 4.10). To achieve this, we implemented an abstract solver class, and the CG and the preconditioners as children classes (see Listing 4.10). The *ConjugateGradient* class contains to special methods: *cg()* which abstracts the CGM and PCGM algorithms, and receives the desired preconditioner as an argument (run only the CGM solver we pass the *identity* preconditioner). The second special method is *fcg()* which abstracts the FCG algorithm.

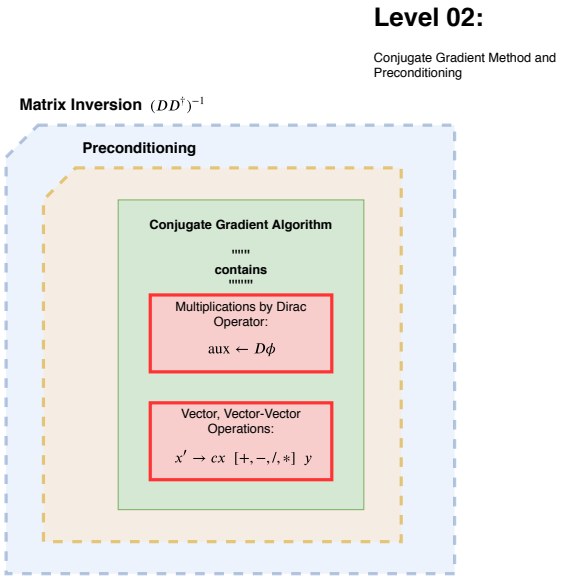

Figure 4.9: Matrix Inversion $(DD)^{-1}$ via PCGM and FCG

```
class Solver
{
    public:
      Solver() {}
      virtual ~Solver() {}
      virtual void apply_preconditioning( ... ) = 0;
      virtual const std::string name () const = 0;
};
class ConjugateGradient: public Solver
{     ...
    public:
      void apply_preconditioning( ... )
      ...
      void cg(..., Solver& precond);
      void fcg(..., Solver& precond);
      ...
};
class Jacobi: public Solver
```

```
{       ...
    public:
      void apply_preconditioning( ... )
      ...
};
class InvertedDiagonal: public Solver
{       ...
    public:
      void apply_preconditioning( ... )
      ...
};
class Identity: public Solver
{       ...
    public:
      void apply_preconditioning( ... )
      ...
};
class SSOR: public Solver
{       ...
    public:
      void apply_preconditioning( ... )
      ...
};
```

Listing 4.10: Class herarchy for solver-preconditioner classes.

We tested the implementation by generating a field filled with complex gaussian random numbers and applying the $DD^\dagger(\sigma)$ operator to it, then the result and the matrix are used as inputs for the algorithm and the solution is then compared to the original field (see Listing 4.11). We implemented a struct *cg manager* to avoid the allocation an reallocation of auxiliary buffers in repeted calls of the solver, creating and destroying this buffers only once during all the application running. This struct contains the CG function, along with some other tools to follow the trail of convergence properties.

```
// Main Solver declaration.
ConjugateGradient          conjGrad_solver{L, tolerance};

// Preconditioners declarations
Identity                   identity_precond;
InvertedDiagonal           invDiag_precond;
Jacobi                     jacobi_precond{L, tolerance, 4};
ConjugateGradient          conjGrad_precond{L, tolerance, 4};
SSOR                       ssor_precond{L};

// Aplication of the same solver with different preconditioners
conjGrad_solver.precond_cg(D, input, output_I, identity_precond);
//                                              ^^^^^^^^^^^^^^^^
conjGrad_solver.precond_cg(D, input, output_InvDiag, invDiag_precond);
//                                                   ^^^^^^^^^^^^^^^
```

```
conjGrad_solver.fcg(D, input, output_jc, jacobi_precond);
//                                       ^^^^^^^^^^^^^^
conjGrad_solver.fcg(D, input, output_cg, conjGrad_precond);
//                                       ^^^^^^^^^^^^^^^^
conjGrad_solver.precond_cg(D, input, output_ssor, ssor_precond);
//                                                ^^^^^^^^^^^^

// Checking results
check_I_cg                  = (reference == output_I);
check_InvDiag_cg            = (reference == output_InvDiag);
check_jc_cg                 = (reference == output_jc);
check_cg_cg                 = (reference == output_cg);
check_ssor_cg               = (reference == output_ssor);
```

Listing 4.11: Testing inversion with different preconditioners.

Finally include a testing module to check for the correctness of these operations:

```
static char *all_tests()
{
  my_run_test(test_identity_cg);
  my_run_test(test_InvDiag_cg);
  my_run_test(test_jc_cg);
  my_run_test(test_cg_cg);
  my_run_test(test_ssor_cg);
  return 0;
}
```

Listing 4.12: Solver testing.

4.2.6 Convergence rate comparison of the different preconditioners

In Figure 2 we compare effect of the different preconditioners in convergence behavior by plotting the residual $r_i = |\mathbf{b} - DD^{\dagger}(\sigma)\mathbf{x}_i|$ against the number of iterations. Here σ corresponds to the scalar field, which in this case was gaussian-randomly generated with mean value 0 and standard deviation 1. In Figure 1 (a) We compare Identity (dummy), Inverse Diagonal, CG and HSOR. In (a) Jacobi preconditioner is not plotted because it actually increases the number of iterations needed for convergence. This is because the used scalar field makes $DD^{\dagger}(\sigma)$ to be not diagonally dominant (A requirement for Jacobi and other iterative methods to work properly). If we shift the mean value of the scalar field in such a way $DD^{\dagger}$ became diagonally dominant, we find not only that the Jacobi method starts working better but the overall ratio of convergences of all the other methods improves drastically. In Figure 2 (b) we plot $\mathbf{r}_i$ vs number iterations for all the preconditioners for a matrix $DD^{\dagger}(\sigma)$ created with a scalar field with mean value 5 and a lattice size of 32. In this case we see how the convergence is way more faster in all cases, including Jacobi.

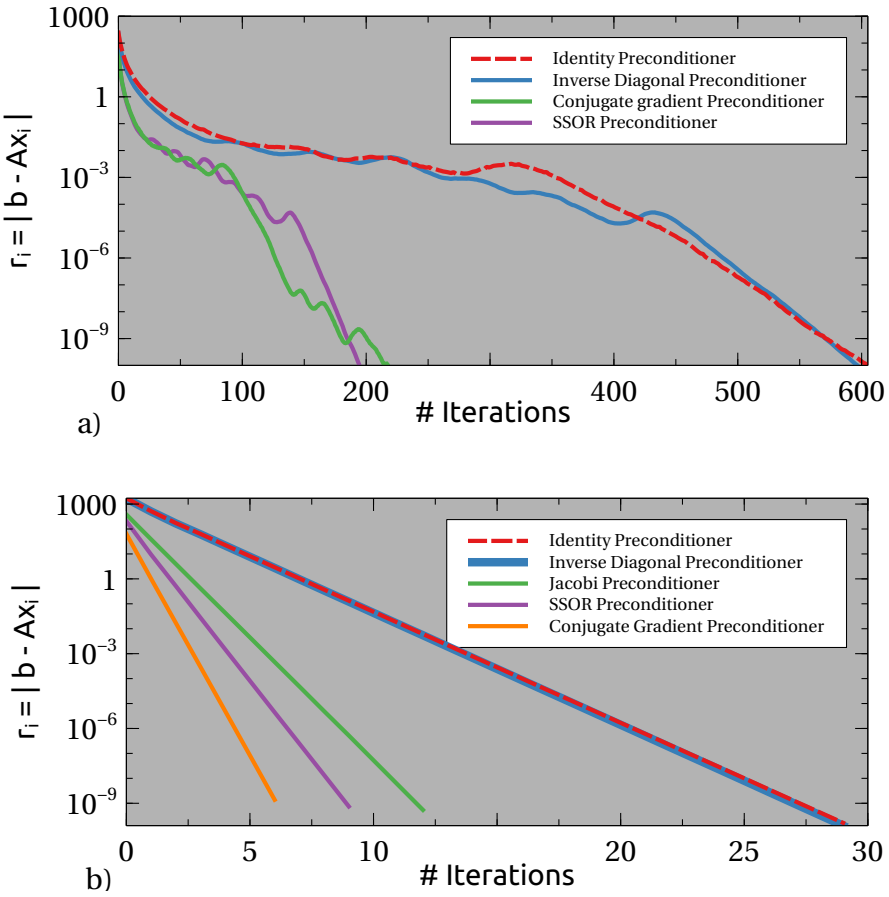

Figure 4.10: (a) Convergence comparison of Idendity preconditioner, Inverse Diagonal preconditioner, Conjugate Gradient preconditioner and SSOR preconditioner for a $L = 18$ and mean value of gaussian-randomly generated scalar field equal zero. (b) Convergence comparison of Idendity preconditioner, Inverse Diagonal preconditioner, Jacobi preconditioner, Conjugate Gradient preconditioner and HSOR preconditioner for a $L = 32$ and mean value of gaussian-randomly generated scalar field equal 5.

Benchmarking

In order to compare the performance of our different implementations for the matrix inversors, we plot the execution time of the solvers using our two preferred types of multiplication (homemade On-The-Fly and MKL Sparse) against the number of OMP/MKL threads for lattices of $L = 24$ and $L = 32$, using the HSOR preconditioner. As we can see below, in the Figure 4.11, and as expected from our time measurements for the *matrix-vector* operations, the Sparse implementation happen to be roughly one order of magnitude faster than the homemade counterparts.

We also note that the lack of scaling of the version using sparse multiplications indicates that the best strategy for parallelization is the use of few threads per simulation, and dedicating more threads or processes for totally independent simulations.

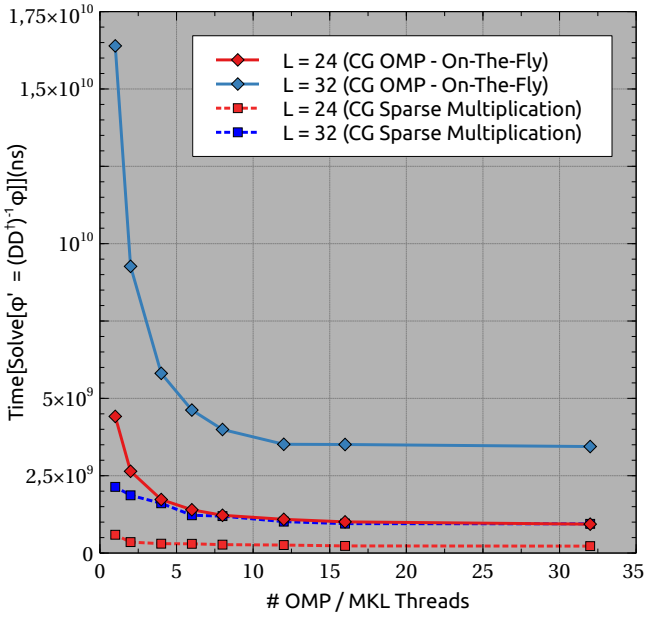

Figure 4.11: Execution times of Preconditioned Conjugate Gradient (HSOR) for field sizes $L = 32, 64$ and for $4, 8, 16, 32$ mpi-processes/threads.

4.3 Level 03: Using $[DD^{\dagger}(\sigma)]^{-1}$: Pseudofermion Sampling and Force Calculation

At this level we consider to main procedures:

4.3.1 Pseudofermion Sampling

The aim of this algorithmic level is to sample the distribution $e^{-\phi^{\dagger}(DD^{\dagger})^{-1}\phi}$, sampling first a gaussian distribution of the form $\exp(-\chi^{\dagger}\chi)$ using the Marsaglia polar method [20] (a standard C++ implementation in Listings 4.13), to finally use the trick of writing $\phi^{\dagger}(DD^{\dagger})^{-1}\phi = [\phi^{\dagger}(D^{\dagger})^{-1}][D^{-1}\phi] = [D^{-1}\phi]^{\dagger}[D^{-1}\phi] \rightarrow \phi = D\chi$ so we obtain the original distribution. The algorithmic structure is shown in Figure 4.12.

```
1  void polar(double *x1, double *x2)
2  {
3    double u, v, q, p;
4
5    do
6    {
7      u = 2.0 * random() - 1;
8      v = 2.0 * random() - 1;
9      q  = u * u + v * v;
10   }
11   while (q >= 1.0 || q == 0.0);
12
13   p = sqrt(-2 * log(q) / q);
```

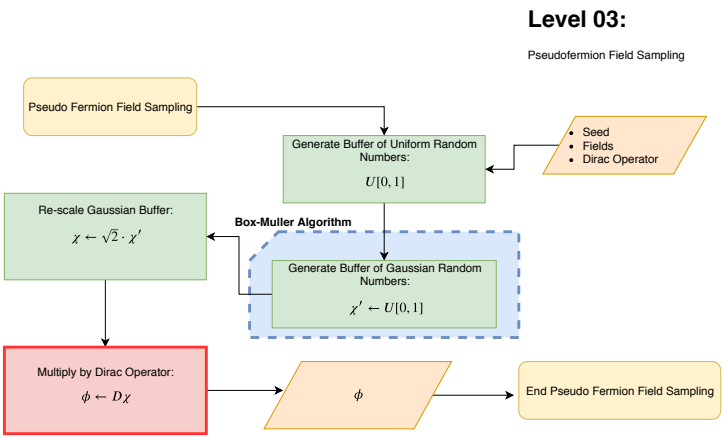

Figure 4.12: Pseudofermion Sampling

```
14    *x1 = u * p;
15    *x2 = v * p;
16 }
```

Listing 4.13: Simple C++ implementation of Marsaglia polar method

4.3.2 Pseudofermion Force Calculation

Here we have one inversion (the other one is repeated), performed using Preconditioned Conjugated Gradient or Flexible Conjugate Gradient. The algorithmic structure, as described in 3.1.5, is shown in Figure 4.13.

4.4 Level 04: Using Forces: Molecular Dynamics Simulation

4.4.1 Solving Canonical Equations of Motion: Molecular Dynamics

Note that if the trajectories described by the equations 3.5 were simulated exactly then $\mathcal{H}(\mathbf{x}, \mathbf{p})$ would be, of course, a constant of motion. In this scenario the movement would be always over a hypersurface of constant probability density, and 3.8 would lead always to an acceptance probability of 1. Nevertheless, in practice equations 3.5 cannot be solve exactly and we will end up with a finite difference $|\Delta\mathcal{H}|$. The key point is that, the exact our trajectories are, the smaller is $|\Delta\mathcal{H}|$ (and higher or acceptance rates). This is something that we can always achieve with arbitrary precision if we take an important and subtle detail into account: Not all numerical integrators (such as the Euler, mid point, or Runge-Kutta methods) preserve phase-space measure (a feature that is required in order to ensure HMC works properly as an MCMC algorithm). Fortunately, there are special algorithms designed to meet this preservation: the so called **sympletic integrators** . This type of integrator are extensively use in Molecular Dynamics simulations, or in any other application where any type of

==

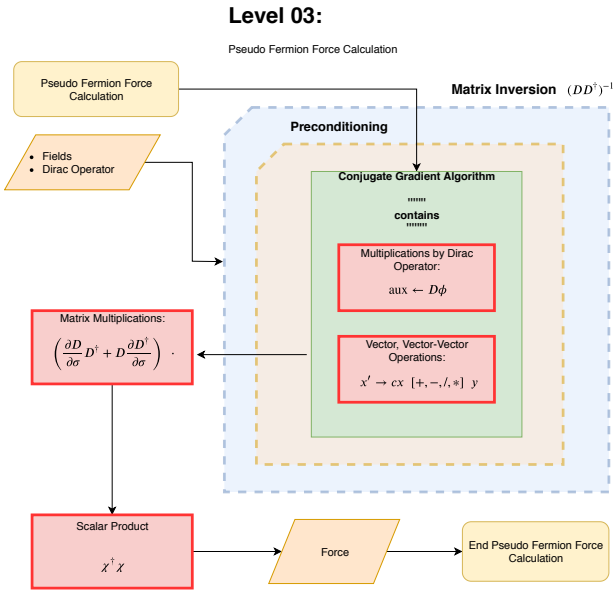

Figure 4.13: Pseudofermion Force Calculation.

Newtonian/Hamiltonian dynamics should be computed for large trajectories. These type of approaches have the following characteristics:

- Error goes as the square of time-step: $|\Delta H| \sim O(\delta t^2)$.
- They conserve phase-space measure.

For this project, we have chosen to use the Stormer-Verlet Integrator, which is highly reliable and relatively easy to implement.

Stormer-Verlet Integrator

With this methodology we define the **micro-evolution operators** $I_p(\delta t)$ and $I_\phi(\delta t)$, for p and ϕ respectively, and up to first order, are given by [30]:

$$I_p(\delta t)\ (p_{\mathbf{x}}^{(i)}, \phi_{\mathbf{x}}^{(i)}) = \left(p_{\mathbf{x}}^{(i)} - \nabla_\phi H(p_{\mathbf{x}}^{(i)}, \phi_{\mathbf{x}}^{(i)})\delta t, \phi_{\mathbf{x}}^{(i)}\right) = (p_{\mathbf{x}}^{(i+1)}, \phi_{\mathbf{x}}^{(i)}), \tag{4.37}$$

$$I_\phi(\delta t)\ (p_{\mathbf{x}}^{(i)}, \phi_{\mathbf{x}}^{(i)}) = \left(p_{\mathbf{x}}^{(i)}, \phi_{\mathbf{x}}^{(i)} + \nabla_p H(p_{\mathbf{x}}^{(i)}, \phi_{\mathbf{x}}^{(i)})\delta t\right) = (p_{\mathbf{x}}^{(i)}, \phi_{\mathbf{x}}^{(i+1)}). \tag{4.38}$$

In terms of $I_p(\delta t)$ and $I_\phi(\delta t)$ the full time-evolution operator given by the Stormer-Verlet integrator is written down as:

$$(p^{(n)}, \phi^{(n)}) = \left[I_\phi\left(\frac{\delta t}{2}\right) I_p(\delta t) I_\phi\left(\frac{\delta t}{2}\right)\right]^n (p^{(0)}, \phi^{(0)}), \tag{4.39}$$

where n is the number micro trajectories, and $n\delta t$ is the total simulation time. Finally we can obtain a explicit version of this expression by applying ∇_p and ∇_ϕ directly to our discretized Hamiltonian:

$$p_{\mathbf{x}} = \nabla_p H(p_{\mathbf{x}}, \phi_{\mathbf{x}}) \quad ; \quad F_{\mathbf{x}} := -\nabla_\phi H(p_{\mathbf{x}}, \phi_{\mathbf{x}}). \tag{4.40}$$

The relation $\delta t = \tau/n$ should be tunned in such a way that $|\Delta H|$ is not too big, nor too small. If $|\Delta H| \approx 0$ then the evolution of the Markov Chain is prone to strong autocorrelations (small trajectories) and the sampling of the phase space becomes inefficient. In the other hand if $|\Delta H| >> 0$ then the acceptance rate decreases and the convergence of the algorithm slows down.

Here at this level we have the internal Molecular Dynamics simulation. It is made up from a couple of Action updates, a couple of energy measurements, and a Stormer-Verlet integrator, as described in 4.4.1. The algorithmic structure is shown in Figure 4.14.

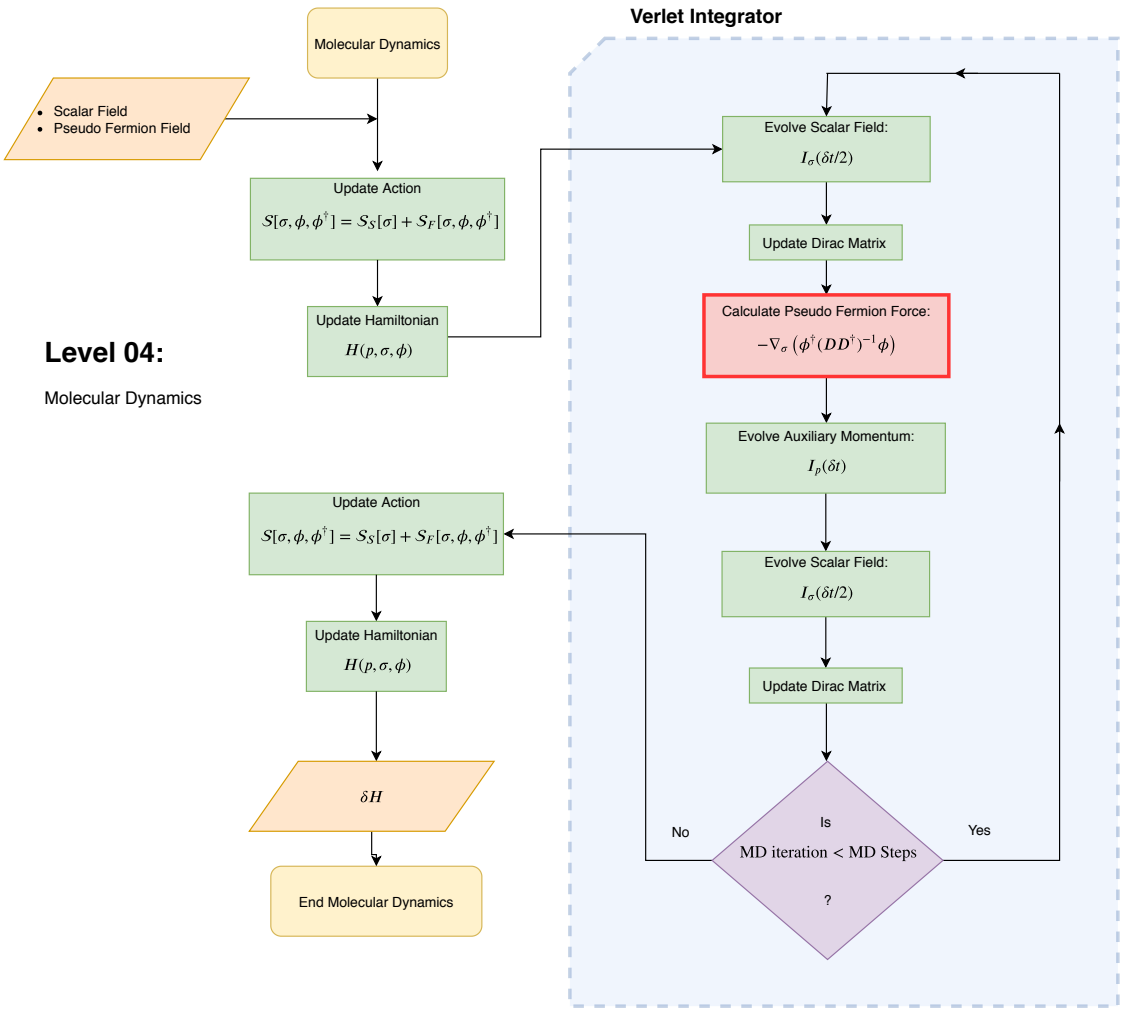

Figure 4.14: Molecular Dynamic Simulation.

4.5 Level 05: Including Molecular Dynamics: Hamiltonian Monte Carlo Simulation.

This level consists in the actual HMC simulation. The algorithmic structure, as described in 3.1.5, is shown in Figure 4.15.

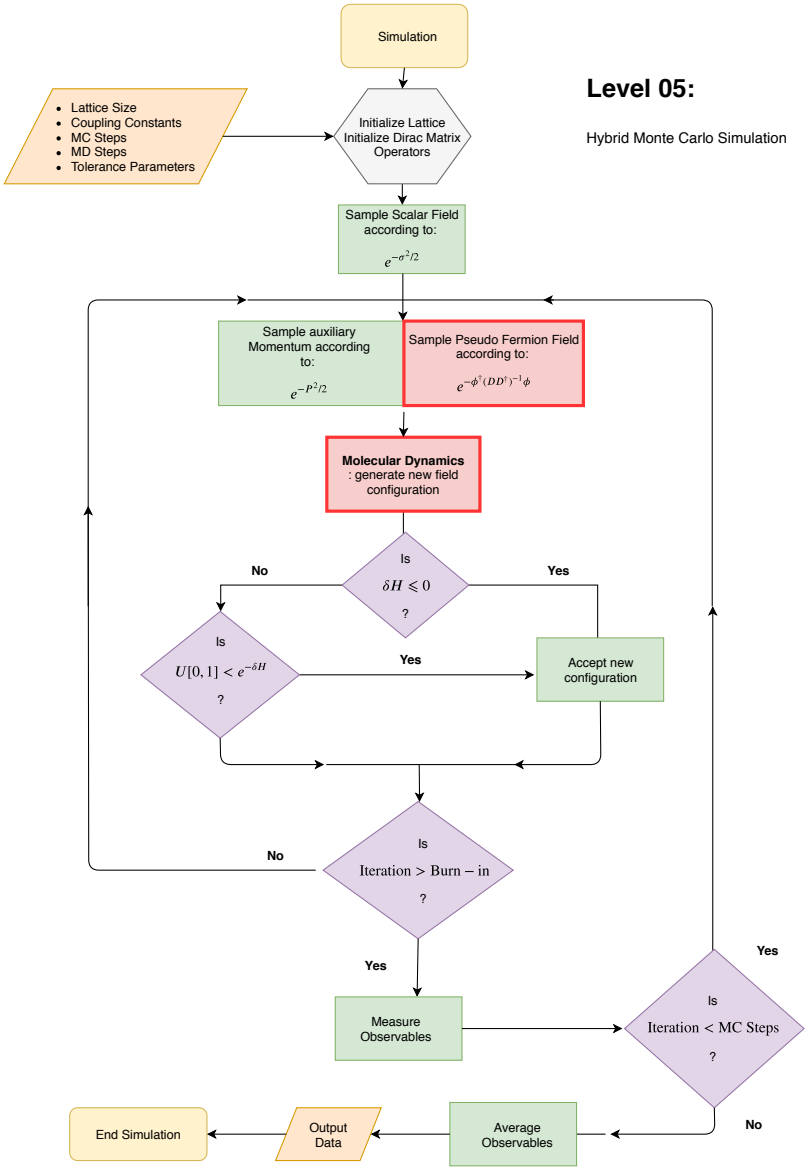

Figure 4.15: Hamiltonian Monte Carlo Simulation.

Benchmarking

Now we compare the performance for full single simulations, once again we plot the execution time using our two preferred types of multiplication (homemade On-The-Fly and MKL Sparse) inside the solver routine (see Figure 4.16), against the number of OMP/MKL threads for lattices of $L = 12, 18, 24, 28, 32, 36$. Again we conclude that the Sparse version overpasses the homemade

versions and is our final preferred choice. After this point, and due to the poor scaling of the Sparse method, we reaffirm our final strategy of dedicating more processes to independent simulations than to the internal parallelization of our algorithms.

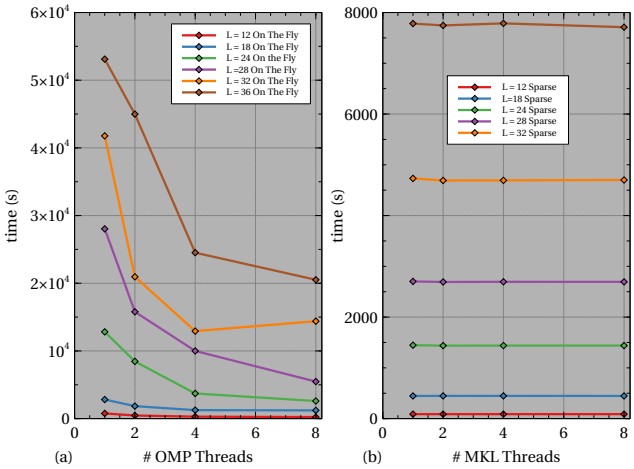

Figure 4.16: Running time for a entire simulations with $L = 12, 16, 18, 24, 32, 36$ with number of threads $T = 1, 2, 4, 8$ for (a) non optimized On-The-Fly approach (b) MKL Sparse Matrix approach. (Note 8000 s $\approx$ 2h 13 min).

4.6 Level 06: Many HMC simulations: Parameter sweeping and data analysis.

This section consists, first, of a sweep of parameters: one over the different lattice sizes (scale transformations), and another over the ranges of coupling constants. The second part corresponds to the analysis of the data produced by these sweeps, and its objective is to determine critical points and critical exponents, among other properties. The algorithmic structure is shown Figure 1.

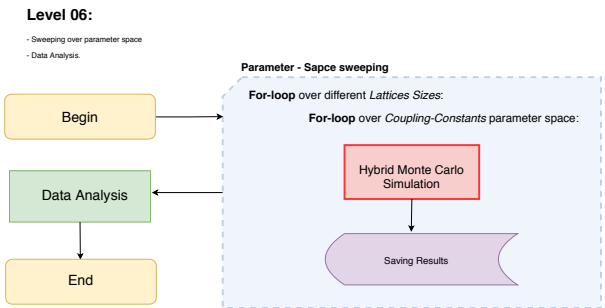

Figure 4.17: Parameter sweeping and data analysis.

Benchmarking

At this level we target a trivial MPI parallelization in which we lunch multiple identical and totally independent simulations, each with different coupling values over the parameter space. Strong and weak scaling benchmarks are shown in Figure 4.18 for a lattice of $L = 32$ (a) and (b). In this approach the communication times are non-existent and the scaling is almost linear as seen in Figure 4.18 (a) inset.

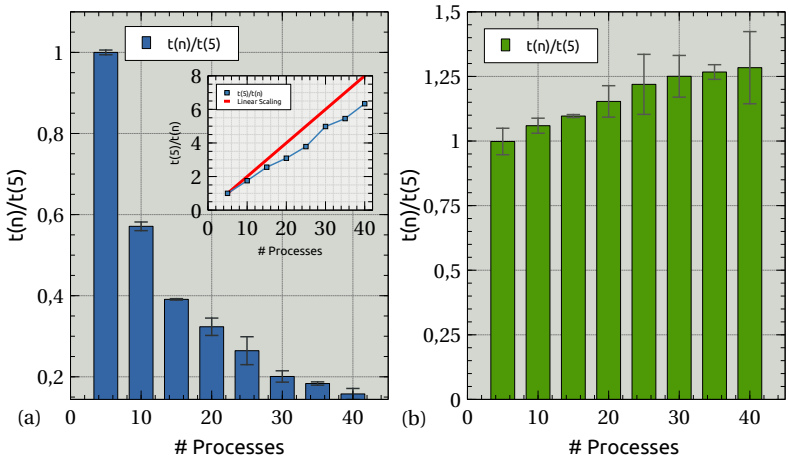

Figure 4.18: (a) [inset] Strong and (b) Weak scaling for running independent full simulations for $5, 10, 15, 20, 25, 30, 35, 40$ MPI processes and $L = 32$.

Chapter 5

Results

In this chapter we briefly discuss physical results focused on critical phenomena extracted from our simulations for the ϕ^4 model. The corresponding analysis for the GNY model will be left as future work.

5.1 Testing Molecular Dynamics algorithm

Before moving on to the analysis of actual Hamiltonian Monte Carlo simulations, it is important to check the correct functioning of the main engine to propose new configurations: the molecular dynamics algorithm. We go through this showing that its results and behavior are consistent with what we expect on numerical and physical grounds

The two crucial properties our molecular dynamics algorithm (Stormer-Verlet) that we might revise are the error scaling of $|\Delta H|$ and the conservation of phase space measure.

5.1.1 $|\Delta H|$ error scaling

In order to check the relation $|\Delta H| \sim O(\delta t^2)$ we plotted the absolute fluctuation of the energy after after carrying out molecular dynamics trayectories of unitary length ($\tau = 1$) , in function of the time step, for the lattices the following sizes $N = 4^3, 8^3, 12^3, 16^3, 20^3, 24^3$ [see Figure 5.1 (a) and Table 5.1].

L	m_L	b_L	R^2
4^3	1.75406e+03	-2.62807e-05	9.9999966e-01
8^3	6.65441e+03	-7.28279e-05	9.9999981e-01
12^3	2.71213e+04	-2.68746e-04	9.9999985e-01
16^3	7.27669e+04	-1.18048e-03	9.9999960e-01
20^3	1.27494e+05	-1.48028e-03	9.9999979e-01
24^3	2.32798e+05	-2.62039e-03	9.9999980e-01

Table 5.1: Linear fitting parameters for $|\Delta H|$ vs δt^2, with the lattices sizes $N = 4^3, 8^3, 12^3, 16^3, 20^3, 24^3$.

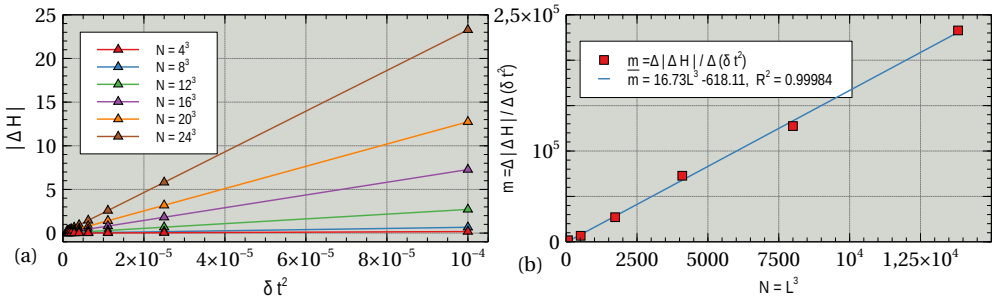

Figure 5.1: (a) Energy error behavior with δt^2 for lattice sizes $N = L^3 = 4^3, 8^3, 12^3, 16^3, 20^3, 24^3$. (b) Behavior of the slope $m = \Delta|\Delta H|/\Delta(\delta t^2)$ as function of L^3.

In Table 5.1 we can see that the squared linear correlation coefficient is ~ 1 for all the cases. Furthermore, we can verify that all the intercepts b_L are small (~ 0) which goes in accordance with the expected exact behavior of the solutions in the continuous-time limit. Regarding the behavior of the slopes [see Figure 5.1 (b)], we found out that they show a linear relationship with L^3, which is in accordance with the fact that H grows in contributions as L^3.

5.1.2 Conservation of Phase Space Measure

If the measure is conserved then we should expect $\langle \exp(-\Delta H) \rangle = 1$. To see this we write:

$$\langle \exp(-\Delta H) \rangle = \frac{1}{\mathcal{Z}} \int D[P,\phi] \exp(-H[P,\phi]) \exp(-\Delta H[P',\phi',P,\phi]) \tag{5.1}$$

Changing variables $(P,\phi) \to (P',\phi')$ we get

$$= \frac{1}{\mathcal{Z}} \int D[P',\phi'] \left| \frac{\partial(P,\phi)}{\partial(P',\phi')} \right| \exp(-H[P',\phi']), \tag{5.2}$$

now, if the measure is actually conserved, then $|\partial(P,\phi)/\partial(P',\phi')| = 1$ and 5.2 is evaluated to 1. To test this the we compute $\langle \exp(-\Delta H) \rangle$ for different values of the coupling constant κ, for the lattices sizes $N = 4^3, 8^3, 12^3, 16^3, 20^3, 24^3$ [see Figure 5.2].

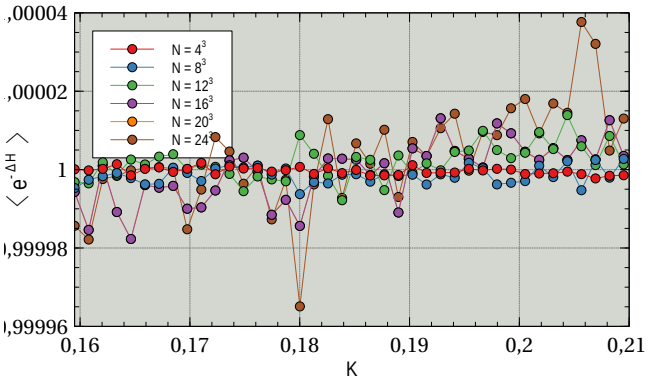

Figure 5.2: Computation of $\langle \exp(-\Delta H) \rangle$ for $\kappa \in [0.16, 0.21]$, with the lattices sizes $N = 4^3, 8^3, 12^3, 16^3, 20^3, 24^3$.

From Figure 5.2 we can see that for all lattice sizes this expected value is accurate up to the fifth decimal digit. A tendency to increasing the errors is observed with the growth of the lattice size.

5.2 Error Analysis using the Jackknife method

Due to the numerical nature of our observable calculations, all values obtained following our Monte Carlo methodology will be accompanied by statistical errors. A practical way to calculate these errors is the use of a methodology called the Jackknife method [19], specially useful to correct bias estimations in sampling sampling procedures [9].

With the Jackknife method we divide our chain of configuration into bins of width w, so if our original set is given by $\{x^{(1)},\ldots,x^{(N)}\}$, our bins, or sub-sets or width w are going to be given by: $\{x^{(1)},\ldots,x^{(w)}\},\{x^{(w+1)},\ldots,x^{(2w)}\},\ldots,\{x^{(n-1)w+1},\ldots,x^{(nw)}\}$, where n is the total number of bins ($N = nw$).

If $\tilde{O}^{(k,w)}$ corresponds to the average of an observable over the $k-$bin, *i.e.* if

$$O^{(\tilde{k},w)} = \frac{1}{w}\sum_{i\in k-bin} O(x^{(j)}), \tag{5.3}$$

and $\bar{O}$ is the average over the whole set:

$$\bar{O} = \frac{1}{n}\sum_{k}\tilde{O}^{(k,w)}, \tag{5.4}$$

the the expression

$$\delta_w = \sqrt{\frac{1}{n(n-1)}\sum_{k}\left(\tilde{O}^{(k,w)}-\bar{O}\right)^2} \tag{5.5}$$

is the standard error obtained by treating $\tilde{O}^{(k,w)}$ to be independent samples. If we set $w = 1$ then $\delta_1 = \sigma_0$ is an estimator of the square root of the variance, meaning then that $\pm\sigma_0/\sqrt{N}$ is our statistical errors. In this way we write down our final results as

$$O = \bar{O} \pm \frac{\sigma_0}{\sqrt{N}}. \tag{5.6}$$

5.3 Computation of Observables

We ran parallel simulations to compute vacuum expectation values for the magnetization $m = V^{-1}\sum_{\mathbf{x}}\phi_{\mathbf{x}}$, magnetic suscetibility $\chi = \langle M^2\rangle - \langle M\rangle^2$ and Binder cumulant $U = \langle M^4\rangle/(\langle M^2\rangle)^2$ as functions of the coupling constant $\kappa \in [0.16, 0.21]$ on $2+1$ lattice space-times for the lattices sizes $N = 4^3, 8^3, 12^3, 16^3, 20^3, 24^3$. The Markov Chain produced was of length 2x10^4, 10^3 of which were used for the thermalization of the system and ignored in the collection of statistics (burn-in time). Each Markov link required a whole Molecular Dynamics simulation to be produced, each of then of a 2x10^2 steps. The value of the coupling constant λ was chosen to be 1.1x10^0 according to Hasenbusch [13] suggestion.

Analyzing the behavior of m and χ with κ [see Figure 5.3 (a) and (b)] we observe the occurrence of a phase transition somewhere between $\kappa = 0.18$ and $\kappa = 0.19$. To spot out the critical value κ_c, the corresponding diagram of the Binder cumulant is used [see Figure 5.4 (a)]: the point of intersection of the U curves for the different lattices is found at $\kappa_c = 0.18625$, which differs from the value reported by [13] ($\kappa_c = 0.18644$) with a margin of 0.1%.

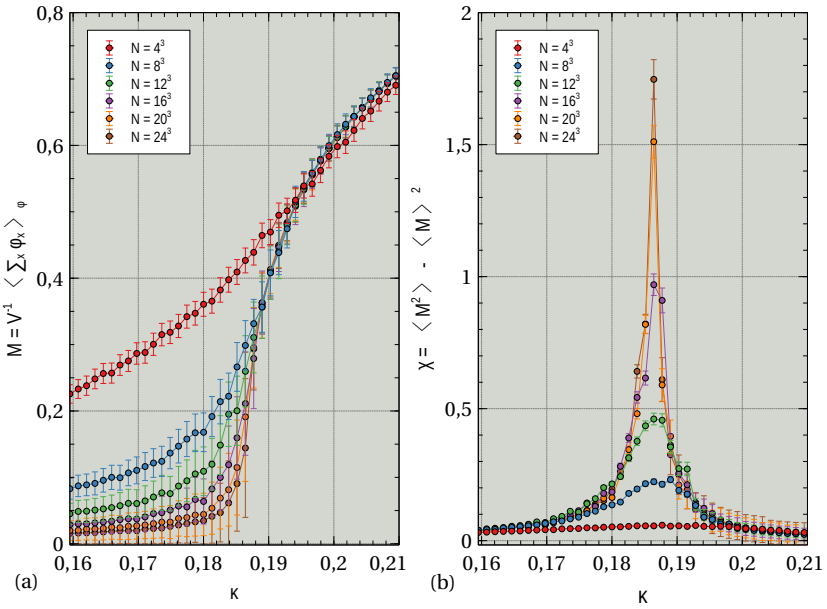

Figure 5.3: (a) Magnetization $m = V^{-1}\langle \sum_{\mathbf{x}} \phi_{\mathbf{x}} \rangle_{\phi}$ and (b) Magnetic Susceptibility $\chi = \langle M^2 \rangle - \langle M \rangle^2$for $\kappa \in [0.16, 0.21]$, with the lattices sizes $N = 4^3, 8^3, 12^3, 16^3, 20^3, 24^3$. Error bars calculated using jackknife method.

5.4 Critical Phenomena

One of the features of second order phase transition is that near its vicinities several quantities diverge. It turns out to be extremely important for the theory of critical phenomena to understand and characterize the specific way in which these quantities diverge. This characterization is done by defining what is know as critical exponents [32]. To see where do these critical exponent appear, and to discuss later on their importance, let us consider first an arbitrary divergent observable $F(t)$, where t is some parameter whose variation drives on the phase transition (for example the reduced temperature $t = (T - T_c)/T_c$ in a ferromagnetic system) . One might expect that near the critical point $F(t)$ behave as:

$$F(t) = A|t|^{\lambda}(1 + bt^{\lambda_1} + \dots). \tag{5.7}$$

The quantity λ is a critical exponent and it has the striking property of being *universal*. A quantity is universal if it assumes exactly the same value for any system within a given *universality class*. A universality class is characterized by the dimension of the system, the range of the interaction and the symmetry of the order parameters [6].

In the cases of lattice models a very usual technique used for the computation of critical exponent is known as finite-size scaling (FSS) [6]. Using this approach the the critical exponents are then extracted from the scaling of the observables with the lattice size, for instance, near the critical point the magnetic susceptibility behaves as:

$$\chi = aL^{2-\eta}(1+bL^{-\omega}+cL^{-\omega'}+dL^{-2\omega}+\cdots)+B \tag{5.8}$$

where B is an analytic background and L the linear size of the p.b.c lattice [14]. For $\partial\bar{U}/\partial\kappa|_{\kappa_c}$, the behavior is given by:

$$\partial\bar{U}/\partial\kappa|_{\kappa_c} = aL^{1/\nu}(1+bL^{-\omega}+cL^{-\omega'}+dL^{-2\omega}+\cdots). \tag{5.9}$$

η exponent

Following [13] [2] [14] and using the following ansatz suggested by 5.8:we obtain the critical exponent η for Binder cumulant fixed at $\bar{U} = 1.6032$ (value at estimated phase transition), and with $\lambda = 1.1$.

$$\bar{\chi}(L) = c + dL^{2-\eta} \tag{5.10}$$

[see Figure 5.4 (b)]. The found value was $\eta = 0.02921$ which differs from the value reported by [13] ($\eta = 0.03357$) in 12.98% [13].

ν exponent

Also following [13] [2], we calculate the critical value ν for the slope of the Binder cumulant at $\bar{U} = 1.6032$ (value at estimated phase transition) for $\lambda = 1.1$. The scaling behavior for $\bar{U}$ was fitted using a simple power law ansatz suggested by 5.9 [see Figure 5.4 (c)] as:

$$\frac{\partial\bar{U}}{\partial\kappa} = cL^{1/\nu} \tag{5.11}$$

The found value was $\nu = 0.71181$ which differs from the value reported by [13] ($\nu = 0.6289$) in 14.185%.

5.5 Correlation Time and Critical Slowing Down

To study autocorrelations in our Markov chains we proceed to measure the observable $G_c(t) = \langle\phi(t_0)\phi(t_0+t)\rangle$. For instance in Figure 5.5 (a) we show $G_c(t)$ as a function of t for a lattice $L = 12$ and $\lambda = 1.145$ and $\kappa = 0.18055$. Our data are fitted as $\propto e^{-t/\xi}$ (red dashed line), from which we can extract the correlation time ξ (In this particular case we found $\xi = 28,57$ in units of Monte Carlo time). This correlation time naturally gives us the distance parameter between configurations in our chain that we must use to extract sufficiently uncorrelated configurations for further use in statistical calculations of observables. To see how the autocorrelation time depends on the physical system, we plot the correlation time ξ for a lattice $L = 12$ as a function of κ in the interval $[0.1595, 0.2095]$ (which

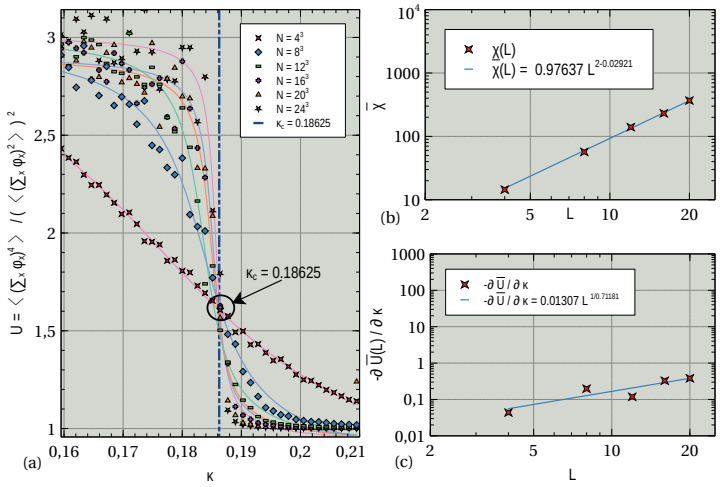

Figure 5.4: (a) Binder Cumulant $U = \langle M^4 \rangle / (\langle M^2 \rangle)^2$ for $\kappa \in [0.16, 0.21]$, with the lattices sizes $N = 4^3, 8^3, 12^3, 16^3, 20^3, 24^3$. (b) $\bar{\chi}(L) = dL^{2-\eta}$ fitting at $\lambda = 1.1$ (c) $\frac{\partial \bar{U}}{\partial \kappa} = cL^{1/\nu}$ at $\lambda = 1.1$.

contains κ_c), and setting $\lambda = 1.145$. We can clearly see that, in the vicinity of the critical point (see Figure 5.5 (b)), the correlation time sharply increases. This increase in autocorrelation time implies that if we want to extract a fixed number of uncorrelated configurations from our Markov chain for statistical calculations, we must necessarily increase the total length of the chain, which is further translated in a enhancement of the required computational effort of the simulations in the parameter regions close κ_c. This phenomenon is called as *critical slowing down* and is well known in the context of Monte Carlo simulations, where is typically observed near the critical points of a theory [26] [27]. The severity of the critical slowing down depends on the algorithm used and on the observable one is analyzing. Its danger is that, if it is not taken into account, the autocorrelation time can easily become much larger than the prefixed in Monte Carlo time, involving the generation of Markov states without any statistical value.

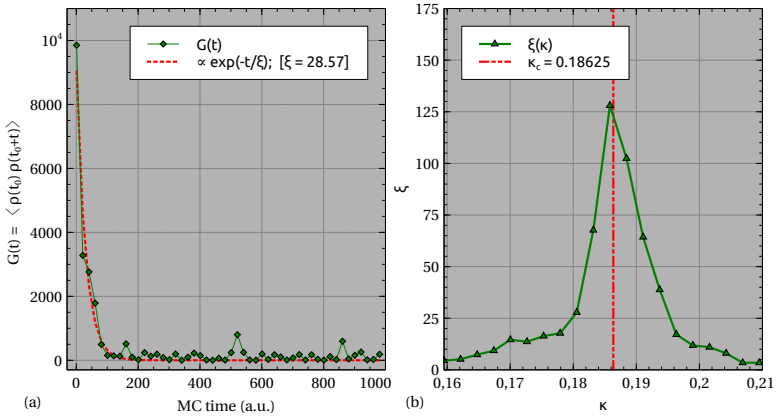

Figure 5.5: (a) $G_c(t) = \langle \phi(t_0)\phi(t_0+t) \rangle$ as a function of t for a lattice $L = 12$ with $\lambda = 1.145$ and $\kappa = 0.18055$ (b) correlation time ξ for a lattice $L = 12$ as a function of κ in the interval $[0.1595, 0.2095]$ with $\lambda = 1.145$.

Chapter 6

Conclusion

In this project we created from scratch software to simulate 3–dimensional ϕ^4 and 3-dimensional Gross-Neveu-Yukawa models on the lattice, using Monte–Carlo methods. The purpose of this work was implement the set of techniques traditionally used in lattice QCD simulations and adapt them for the use in more simple theories such as ϕ^4 and GNY, targeting as specialization the study of critical phenomena. The final aim of this proposal was to obtain insight in these later theories, and to get into contact with the computational and algorithmic complications associated with simulating LQCD: an activity which is widely known to be extremely computationally intensive, and, because of that, a subject of interest for High Performance Computing.

In the development of our implementation we identify matrix-vector operations as the main bottlenecks and the rapidly increasing matrix sizes as our main concern regarding memory limits. We implemented these operations using three different approaches: Full matrix approach using intel mkl cBLAS, sparse matrix approach using intel mkl sparse cBLAS, and finally with a homemade multiplication strategy that generates the matrix entries on computing time (On-The-Fly approach). We checked the consistency of the three methods, disregarding the full matrix or dense approach from the beginning, due to its high memory requirements and relatively slow performance. With the on-the-fly method we proposed and implemented shared memory (OMP) and distributed memory (MPI) parallelizations, requiring the distributed approach the domain decomposition of fields and el use de *Exchange Halos*. After running performance measurements, we conclude that the Sparse version is faster in all cases, besides showing poor scaling in relation to the number of thread used. The On-The-Fly version has the advantage, however, of lacking an upper bound of memory, this because the matrix elements are generated on the multiplication time. This means that it can be used in a wider range of lattice sizes.

After that we implemented de inverse operation $[DD^{\dagger}(\sigma)]^{-1}\phi$ by means of the Conjugate Gradient method in such a way it is compatible with multithreaded versions of the matrix operations previously developed. This along with a preconditioning strategy, we implemented Identity preconditioner, Inverse Diagonal preconditioner, Conjugate Gradient as preconditioner of Flexible Conjugate Gradient, Jacobi method as preconditioner of Flexible Conjugate Gradient, and Hermitian Successive Over-Relaxation.

All these methods analysed and compared in terms of convergence speed-up, being in general HSOR the best of them.

On top of that we constructed Molecular Dynamics layer of our code which intimately rest on the matrix inversion operations. Moreover the MD layer is in turn the engine for the full Hamiltonian Monte Carlo algorithm. At the end we took time measurements for single simulation times at different lattice size scales, both using internal parallelization and trivial parallelization in which we ran many independent simulations with different parameters for the couplings space, being at the end the preferred strategy the one of dedicating more threads or processes for totally independent simulations.

We use our software to extract physics from the well-known ϕ^4 model from which we could find its critical point and its critical exponents, we also analyzed the correlation times for the associated Markov chain and observed the phenomenon of critical slowing down, which due to its characteristics , is of both physical and algorithmic interest.

As future work we leave the complete MPI parallelization of the code, better and more diverse performance measurements, the consideration of algorithmic improvements to attack the problem of critical slowing down and the physical analysis of the GNY model.

Bibliography

[1] D. J. Amit and V. Martin-Mayor. *Field Theory, the Renormalization Group, and Critical Phenomena: Graphs to Computers Third Edition*. World Scientific Publishing Company, 2005.

[2] H. Ballesteros, L. Fernández, V. Martín-Mayor, A. M. Sudupe, G. Parisi, and J. Ruiz-Lorenzo. Critical exponents of the three-dimensional diluted ising model. *Physical Review B*, 58(5):2740, 1998.

[3] R. Barrett, M. W. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*, volume 43. Siam, 1994.

[4] K. Bitar, A. Kennedy, R. Horsley, S. Meyer, and P. Rossi. Hybrid monte carlo and quantum chromodynamics. *Nuclear Physics B*, 313(2):377–392, 1989.

[5] S. Brooks, A. Gelman, G. Jones, and X.-L. Meng. *Handbook of markov chain monte carlo*. CRC press, 2011.

[6] J. Cardy. *Finite-size scaling*. Elsevier, 2012.

[7] R. F. Dashen, B. Hasslacher, and A. Neveu. Semiclassical bound states in an asymptotically free theory. *Phys. Rev. D*, 12:2443–2458, Oct 1975.

[8] S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth. Hybrid monte carlo. *Physics letters B*, 195(2):216–222, 1987.

[9] B. Efron. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics*, pages 569–593. Springer, 1992.

[10] C. Gattringer and C. Lang. *Quantum chromodynamics on the lattice: an introductory presentation*, volume 788. Springer Science & Business Media, 2009.

[11] D. J. Gross and A. Neveu. Dynamical symmetry breaking in asymptotically free field theories. *Physical Review D*, 10(10):3235, 1974.

[12] M. Hanada. Markov chain monte carlo for dummies. *arXiv preprint arXiv:1808.08490*, 2018.

[13] M. Hasenbusch. A monte carlo study of leading order scaling corrections of 4 theory on a three-dimensional lattice. *Journal of Physics A: Mathematical and General*, 32(26):4851, 1999.

[14] M. Hasenbusch. Finite size scaling study of lattice models in the three-dimensional ising universality class. *Physical Review B*, 82(17):174433, 2010.

[15] K. Huang, E. Manousakis, and J. Polonyi. Effective potential in scalar field theory. *Physical Review D*, 35(10):3187, 1987.

[16] M. H. Kalos and P. A. Whitlock. *Monte carlo methods*. John Wiley & Sons, 2009.

[17] L. Kärkkäinen, R. Lacaze, P. Lacock, and B. Petersson. Critical behaviour of the three-dimensional gross-neveu and higgs-yukawa models. *Nuclear Physics B*, 415(3):781–796, Mar 1994.

[18] L. Lellouch, R. Sommer, B. Svetitsky, A. Vladikas, and L. F. Cugliandolo. *Modern Perspectives in Lattice QCD: Quantum Field Theory and High Performance Computing: Lecture Notes of the Les Houches Summer School: Volume 93, August 2009*. OUP Oxford, 2011.

[19] L. Lyons and L. Louis. *A practical guide to data analysis for physical science students*. Cambridge University Press, 1991.

[20] G. Marsaglia and T. A. Bray. A convenient method for generating normal variables. *SIAM review*, 6(3):260–264, 1964.

[21] I. Montvay and G. Münster. *Quantum fields on a lattice*. Cambridge University Press, 1997.

[22] R. M. Neal et al. Mcmc using hamiltonian dynamics. *Handbook of markov chain monte carlo*, 2(11):2, 2011.

[23] M. E. Peskin. *An introduction to quantum field theory*. CRC press, 2018.

[24] W. Rath. *Cluster simulations in the Gross Neveu model*. Humboldt-Universität zu Berlin, 2009.

[25] Y. Saad. *Iterative methods for sparse linear systems*, volume 82. siam, 2003.

[26] S. Schaefer, R. Sommer, and F. Virotta. Investigating the critical slowing down of qcd simulations. *arXiv preprint arXiv:0910.1465*, 2009.

[27] S. Schaefer, R. Sommer, F. Virotta, A. Collaboration, et al. Critical slowing down and error analysis in lattice qcd simulations. *Nuclear Physics B*, 845(1):93–119, 2011.

[28] J. R. Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[29] W. E. Thirring. A soluble relativistic field theory. *Annals of Physics*, 3(1):91–112, 1958.

[30] L. Verlet. Computer" experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical review*, 159(1):98, 1967.

[31] K. G. Wilson. Confinement of quarks. *Physical review D*, 10(8):2445, 1974.

[32] J. M. Yeomans. *Statistical mechanics of phase transitions*. Clarendon Press, 1992.

Appendix A

Appendix

Write your appendix here. Following two are examples.

A.1 The Doubling Problem

To see the appearance of the unwanted poles we must compute the free fermion propagator. For this we calculate the Fourier transform of the Dirac operator for the free case:

$$\begin{aligned} FT[D(\mathbf{m}|\mathbf{n})] = \tilde{D}(\mathbf{p}|\mathbf{q}) &= \frac{1}{|\Lambda|}\sum_{\mathbf{n},\mathbf{m}\in\Lambda} \mathrm{e}^{-i\mathbf{p}\cdot\mathbf{n}a} D(\mathbf{n}|\mathbf{m})\mathrm{e}^{i\mathbf{q}\cdot\mathbf{m}a} \\ &= \frac{1}{|\Lambda|}\sum_{\mathbf{n},\mathbf{m}\in\Lambda} \mathrm{e}^{-i\mathbf{p}\cdot\mathbf{n}a}\left[\sum_{\mu}\gamma^{\mu}\,\frac{\delta_{\mathbf{n}+\hat{\mu},\mathbf{m}}-\delta_{\mathbf{n}-\hat{\mu},\mathbf{m}}}{2a}\right]\mathrm{e}^{i\mathbf{q}\cdot\mathbf{m}a} \\ &= \frac{1}{|\Lambda|}\sum_{\mathbf{n}\in\Lambda}\sum_{\mu}\mathrm{e}^{-i\mathbf{p}\cdot\mathbf{n}a}\frac{\gamma^{\mu}}{2a}\left[\,\mathrm{e}^{i\mathbf{q}\cdot(\mathbf{n}+\hat{\mu})a}-\mathrm{e}^{i\mathbf{q}\cdot(\mathbf{n}-\hat{\mu})a}\right] \\ &= \frac{1}{|\Lambda|}\sum_{\mathbf{n}\in\Lambda}\sum_{\mu}\mathrm{e}^{-i(\mathbf{p}-\mathbf{q})\cdot\mathbf{n}a}\gamma^{\mu}\frac{i}{a}\left[\frac{\mathrm{e}^{i\mathbf{q}_{\mu}a}-\mathrm{e}^{-i\mathbf{q}_{\mu}a}}{2i}\right], \end{aligned} \tag{A.1}$$

but $|\Lambda|^{-1}\sum_{\mathbf{n}\in\Lambda}\mathrm{e}^{-i(\mathbf{p}-\mathbf{q})\cdot\mathbf{n}a} = \delta(\mathbf{p}-\mathbf{q})$, so:

$$\tilde{D}(\mathbf{p}|\mathbf{q}) = \delta(\mathbf{p}-\mathbf{q})\tilde{D}(\mathbf{p}), \tag{A.2}$$

with $\tilde{D}(\mathbf{p}) = i/a\sum_{\mu}\gamma^{\mu}\sin(p_{\mu}a)$. Then, we obtain the momentum-space propagator inverting $\tilde{D}(\mathbf{p})$. To do this we rely in the following identity [2]:

$$\left(m\mathbb{I}+i\sum_{\mu}\gamma^{\mu}b_{\mu}\right)^{-1} = \frac{m\mathbb{I}-i\sum_{\mu}\gamma^{\mu}b_{\mu}}{m^2+\sum_{\mu}b_{\mu}^2}, \tag{A.3}$$

applying this to $\tilde{D}(\mathbf{p})$ we obtain:

$$\tilde{D}^{-1}(\mathbf{p}) = \frac{-ia^{-1}\sum_{\mu}\gamma^{\mu}\sin(p_{\mu}a)}{a^{-2}\sum_{\mu}\sin^2(p_{\mu}a)} \xrightarrow[a\to 0]{} \frac{-i\sum_{\mu}\gamma^{\mu}p_{\mu}}{\mathbf{p}^2}. \tag{A.4}$$

We see that in the continuum limit we obtain the correct pole at $\mathbf{p}=(0,0,0)$ but in the lattice case we observe the presence of 7 extra nonphysical poles $\mathbf{p}=(\pi/a,0,0),(0,\pi/a,0),...,(\pi/a,\pi/a,\pi/a)$.

The Wilson term

The previous unwanted poles are easily removed by adding an extra term to $\tilde{D}(\mathbf{p})$ that vanishes in the $a \to 0$ limit:

$$\tilde{D}(\mathbf{p})_{\alpha,\beta} = \frac{i}{a}\sum_{\mu}(\gamma^{\mu})_{\alpha,\beta}\,\sin(p_\mu a) + \delta_{\alpha,\beta}\frac{1}{a}\sum_{\mu}[1-\cos(p_\mu a)] \tag{A.5}$$

Using the identity A.3 we get:

$$\begin{aligned}\tilde{D}^{-1}(\mathbf{p})_{\alpha,\beta} &= \frac{a^{-1}\sum_\mu[1-\cos(p_\mu a)]\delta_{\alpha,\beta} - ia^{-1}\sum_\mu(\gamma^\mu)_{\alpha,\beta}\sin(p_\mu a)}{a^{-2}\left(\sum_\mu[1-\cos(p_\mu a)]\right)^2 + a^{-2}\sum_\mu\sin^2(p_\mu a)} \\ &\xrightarrow[a\to 0]{} \frac{-i\sum_\mu\gamma^\mu p_\mu}{\mathbf{p}^2}\end{aligned} \tag{A.6}$$

Now both the lattice and the continuum cases share an unique pole at $\mathbf{p} = (0,0,0)$. Now we have to compute the form of this extra term in the space representation, to do so we compute its inverse Fourier transformation:

$$\begin{aligned}D_{\text{wilson}}(n|m)_{\alpha,\beta} &= \text{FT}^{-1}[\delta(\mathbf{p}-\mathbf{q})\tilde{D}_{\text{wilson}}(\mathbf{p})] \\ &= \text{FT}^{-1}\left[\delta(\mathbf{p}-\mathbf{q})\delta_{\alpha,\beta}\frac{1}{a}\sum_\mu[1-\cos(p_\mu a)]\right] \\ &= \frac{1}{|\Lambda|}\sum_{\mathbf{p},\mathbf{q}\in\tilde{\Lambda}} e^{i\mathbf{p}\cdot\mathbf{n}a}\delta_{\mathbf{p},\mathbf{q}}\delta_{\alpha,\beta}\frac{1}{a}\sum_\mu[1-\cos(p_\mu a)]e^{-i\mathbf{q}\cdot\mathbf{m}a} \\ &= \frac{1}{|\Lambda|}\sum_{\mathbf{p}\in\tilde{\Lambda}} e^{i\mathbf{p}\cdot(\mathbf{n}-\mathbf{m})a}\delta_{\alpha,\beta}\sum_\mu\left[\frac{2-e^{ip_\mu a}-e^{-ip_\mu a}}{2a}\right] \\ &= \delta_{\alpha,\beta}\frac{1}{|\Lambda|}\sum_{\mathbf{p}\in\tilde{\Lambda}}\sum_\mu\left[\frac{-e^{i\mathbf{p}\cdot(\hat{\mu}+\mathbf{n}-\mathbf{m})a}+2e^{i\mathbf{p}\cdot(\mathbf{n}-\mathbf{m})a}-e^{i\mathbf{p}\cdot(-\hat{\mu}+\mathbf{n}-\mathbf{m})a}}{2a}\right] \\ &= \delta_{\alpha,\beta}\sum_\mu\left[\frac{-\frac{1}{|\Lambda|}\sum_{\mathbf{p}\in\tilde{\Lambda}}e^{i\mathbf{p}\cdot(\hat{\mu}+\mathbf{n}-\mathbf{m})a}+\frac{1}{|\Lambda|}\sum_{\mathbf{p}\in\tilde{\Lambda}}2e^{i\mathbf{p}\cdot(\mathbf{n}-\mathbf{m})a}-\frac{1}{|\Lambda|}\sum_{\mathbf{p}\in\tilde{\Lambda}}e^{i\mathbf{p}\cdot(-\hat{\mu}+\mathbf{n}-\mathbf{m})a}}{2a}\right],\end{aligned}$$

But knowing that $|\Lambda|^{-1}\sum_{\mathbf{p}\in\tilde{\Lambda}}\exp(i\mathbf{p}\cdot(\mathbf{n}-\mathbf{m})a) = \delta_{\mathbf{n},\mathbf{m}}$ we get:

$$D_{\text{wilson}}(\mathbf{n}|\mathbf{m})_{\alpha,\beta} = -a\sum_\mu\frac{\delta_{\mathbf{n}+\hat{\mu},\mathbf{m}}\delta_{\alpha,\beta}-2\delta_{\mathbf{n},\mathbf{m}}\delta_{\alpha,\beta}+\delta_{\mathbf{n}-\hat{\mu},\mathbf{m}}\delta_{\alpha,\beta}}{2a^2}, \tag{A.7}$$

which, up to order $\mathcal{O}(a^4)$ is equivalent to $-a/2(\partial_\mu)^2$, being this an expression that vanishes in the continnum limit $a \to 0$. Taking into account this result we rewrite our Dirac operator as:

$$\begin{aligned}&D(\mathbf{n}|\mathbf{m})_{\alpha,\beta} \to D(\mathbf{n}|\mathbf{m})_{\alpha,\beta} + D_{\text{Wilson}}(\mathbf{n}|\mathbf{m})_{\alpha,\beta} \\ &= \frac{1}{2}g\sigma\,\delta_{\mathbf{n},\mathbf{m}}\delta_{\alpha,\beta} + \frac{1}{2}\sum_\mu^3(\gamma^\mu)_{\alpha,\beta}\frac{\delta_{\mathbf{n}+\hat{\mu},\mathbf{m}}-\delta_{\mathbf{n}-\hat{\mu},\mathbf{m}}}{2a} - \frac{1}{2}\sum_\mu^3\delta_{\alpha,\beta}\frac{\delta_{\mathbf{n}+\hat{\mu},\mathbf{m}}-2\delta_{\mathbf{n},\mathbf{m}}+\delta_{\mathbf{n}-\hat{\mu},\mathbf{m}}}{2a} \\ &= \frac{1}{2}\left[\frac{3}{a}+g\sigma\right]\delta_{\mathbf{n},\mathbf{m}}\delta_{\alpha,\beta} - \frac{1}{2}\sum_\mu^3\frac{(\mathbb{I}-\gamma^\mu)_{\alpha,\beta}\delta_{\mathbf{n}+\hat{\mu},\mathbf{m}}+(\mathbb{I}+\gamma^\mu)_{\alpha,\beta}\delta_{\mathbf{n}-\hat{\mu},\mathbf{m}}}{2a}.\end{aligned} \tag{A.8}$$

Finally we write down this in a more compact way by defining $\gamma_{-\mu} = -\gamma_\mu$:

$$D(\mathbf{n}|\mathbf{m})_{\alpha,\beta} = \frac{1}{2}\left[\frac{3}{a}+g\sigma\right]\delta_{\mathbf{n},\mathbf{m}}\delta_{\alpha,\beta} - \frac{1}{4a}\sum_{\mu=\pm1}^{\pm3}(\mathbb{I}-\gamma^\mu)_{\alpha,\beta}\delta_{\mathbf{n}+\hat{\mu},\mathbf{m}}. \tag{A.9}$$

A.2 Mattews-Salam formula

For a single fermion flavor the partition function is given by

$$\mathcal{Z}_F^{(f)}[\sigma] = \int \mathcal{D}[\psi^{(f)}, \bar{\psi}^{(f)}] \ \exp\left(\sum_{\mathbf{n},\mathbf{m}\in\Lambda} \bar{\psi}^{(f)}(\mathbf{n}) D(\mathbf{n}|\mathbf{m}) \psi^{(f)}(\mathbf{m}) \right) \tag{A.10}$$

which corresponds to an integral in Grassmann variables. To compute it we use the linear transformation $\Psi^{(f)} = D\psi^{(f)}$. Then using the following grassmannian identity $\eta' = D\eta \rightarrow d\eta' = \det[D] d\eta$ we write:

$$\begin{aligned} \mathcal{Z}_F^{(f)} &= \det[D] \int \mathcal{D}[\Psi^{(f)}, \bar{\psi}^{(f)}] \exp\left(\bar{\psi}^{(f)} \cdot \Psi^{(f)} \right) \\ &= \det[D] \int \mathcal{D}[\Psi^{(f)}, \bar{\psi}^{(f)}] \left(1 + \bar{\psi}^{(f)} \cdot \Psi^{(f)} \right) \\ &= \det[D], \end{aligned} \tag{A.11}$$

where we used the Taylor expansion of the exponential function and the nilpotency property of the Grassmann numbers.

A.3 γ^5-hermiticity of the Dirac operator

To prove this hermiticity we apply γ^5 to both sides of the our Dirac operator:

$$(\gamma^5 D(\mathbf{n}|\mathbf{m}) \gamma^5)_{\alpha,\beta} = \frac{1}{2} \left[\frac{3}{a} + g\sigma \right] \delta_{\mathbf{n},\mathbf{m}} \delta_{\alpha,\beta} (\gamma^5)^2 - \frac{1}{4a} \sum_{\mu=\pm 1}^{\pm 3} \left[\gamma^5 (\mathbb{I} - \gamma^\mu) \gamma^5 \right]_{\alpha,\beta} \delta_{\mathbf{n}+\hat{\mu},\mathbf{m}}, \tag{A.12}$$

and now, from anticonmutation relation $\{\gamma^5, \gamma^\mu\} = 0$ and knowing $(\gamma^5)^2 = I$ we see that:

$$\gamma^5 \gamma^\mu = -\gamma^\mu \gamma^5 \quad \rightarrow \quad \gamma^5 \gamma^\mu \gamma^5 = -\gamma^\mu, \tag{A.13}$$

this implies that:

$$\begin{aligned} (\gamma^5 D(\mathbf{n}|\mathbf{m}) \gamma^5)_{\alpha,\beta} &= \frac{1}{2} \left[\frac{3}{a} + g\sigma \right] \delta_{\mathbf{n},\mathbf{m}} \delta_{\alpha,\beta} - \frac{1}{4a} \sum_{\mu=\pm 1}^{\pm 3} [(\mathbb{I} + \gamma^\mu)]_{\alpha,\beta} \, \delta_{\mathbf{n}+\hat{\mu},\mathbf{m}} \\ &= \frac{1}{2} \left[\frac{3}{a} + g\sigma \right] \delta_{\mathbf{n},\mathbf{m}} \delta_{\alpha,\beta} - \frac{1}{4a} \sum_{\mu=\pm 1}^{\pm 3} [(\mathbb{I} - \gamma^\mu)]_{\alpha,\beta} \, \delta_{\mathbf{n}-\hat{\mu},\mathbf{m}} \\ &= \frac{1}{2} \left[\frac{3}{a} + g\sigma \right] \delta_{\mathbf{n},\mathbf{m}} \delta_{\alpha,\beta} - \frac{1}{4a} \sum_{\mu=\pm 1}^{\pm 3} [(\mathbb{I} - \gamma^\mu)]_{\alpha,\beta} \, \delta_{\mathbf{n},\mathbf{m}+\hat{\mu}} \\ &= D^\dagger(\mathbf{n}|\mathbf{m})_{\alpha,\beta}, \end{aligned} \tag{A.14}$$

where we used $\gamma^{-\mu} = -\gamma^\mu$. With this the demostration is complete.

Our hearts broke without you...,
Our hearts would have followed you...
Our hearts....