An Introduction to Lattice QCD

Riccardo Bianconcini e Francesco Marini

March 6, 2025

Abstract

This work presents an introduction to Lattice Quantum Chromodynamics (LQCD), a non-perturbative numerical approach to solving Quantum Chromodynamics (QCD) on a discretized space-time lattice. We begin by reviewing the fundamental theoretical framework of QCD and the necessity of lattice discretization due to the strong coupling regime at low energies. The core of the report focuses on numerical path integrals, Monte Carlo methods, and different discretization schemes to improve computational accuracy. We also explore the implementation of Monte Carlo evaluations of gluonic path integrals and the Wilson action, laying the groundwork for more advanced studies in lattice gauge theories. The discussion is complemented by detailed code implementations, numerical results, and a comparative analysis of different discretization techniques. All simulations and computations are made available in an open-source GitHub repository [1], ensuring reproducibility and accessibility for further research. This work is heavily inspired by [2].

Contents

List of Figures

Chapter 1

Introduction

The most fundamental theory of the strong interaction, at the time of writing, is **Quantum Chromodynamics (QCD)**, which is a type of Quantum Field Theory called a non-abelian gauge theory, with symmetry group $SU(3)$. In this model, *quarks* are the fundamental particles that make up hadrons, and the force-carrying particles are called *gluons*, which are analogous to photons in QED. The equivalent of the electric charge in QCD is the *color* of the quarks. However, unlike QED, calculating the probability amplitude of any hadronic process occurring at energy scales $\lesssim 1$ GeV cannot be done using the standard perturbative approach. This is because the strong coupling constant, α_s, has a value $\alpha_s \sim 1$ in this energy regime, which is the origin of the term "Strong Force".

To overcome this limitation, scientists have developed an alternative framework for calculating the hadronic spectrum and matrix elements of operators: **Lattice QCD (LQCD)**. Lattice QCD is formulated on a discrete Euclidean space-time grid, where path integrals are employed to compute probability amplitudes. This approach will be analyzed throughout the course of the report.

Lattice QCD has two main advantages:

- The lattice spacing, usually denoted as a, provides an ultraviolet cutoff at π/a, avoiding infinities in the calculations.
- It can be simulated on computers using methods analogous to those applied in Statistical Mechanics.

This report will develop the analysis presented in [2], ensuring a clear exposition of the physical principles behind each topic and providing detailed explanations of the code implementations.

Chapter 2

Numerical Path Integrals

2.1 Discretizing the Path Integral

The quantization of gauge theories, for reasons beyond the scope of this report, is significantly simplified when performed using the path integral formalism, as opposed to the canonical one. Any time we want to evaluate the probability amplitude for a given process, we are actually summing over an infinite number of possible trajectories. In LQCD, we aim to do something similar, with the exception that instead of summing over paths through physical space, we sum over trajectories in the space of field configurations. This is significantly more complicated for two main reasons:

1. Spacetime needs to be discretized to avoid having an infinite number of spacetime points in each configuration.

2. Even after discretization, with a finite number of points, the number of possible field configurations remains extremely large, making it infeasible to evaluate the path integral directly.

Instead of directly starting with the path integral formulation for fields, we start by considering the path integral formulation for point-like particles, as originally introduced by Feynman [3]. We will first present solutions to both problems highlighted above in this specific case and then generalize them to the field configuration.

In order to tackle the first problem, we first need to address is how to represent a path, which is a function $x(t)$, on a computer. This function is completely arbitrary once the endpoints are specified. To prevent the calculations from becoming computationally infeasible, we define the values of $x(t)$ only at the nodes of a discretized time axis. This procedure, called **Time Slicing**, is commonly used to *regularize* the otherwise infinite-dimensional integral.

The path is therefore represented by a vector of numbers $\{x(t_0), x(t_1), \dots, x(t_N)\}$, leading to an Euclidean[1] action of the form:

$$S(x) = \int_{t_i}^{t_f} dt\, L(x, \dot{x}) \quad \rightarrow \quad S_{\text{lat}}[x] \sim a \sum_{j=0}^{N-1} \left[\frac{m}{2} \left(\frac{x_{j+1} - x_j}{a} \right)^2 + \frac{1}{2} \big(V(x_{j+1}) + V(x_j) \big) \right], \tag{2.1}$$

where a is the grid spacing, defined as $a = \frac{t_f - t_i}{N}$. The propagator is then evaluated using the standard formula:

$$K = A \int_{-\infty}^{\infty} dx_1\, dx_2 \dots dx_{N-1}\, e^{-S_{\text{lat}}[x]}, \tag{2.2}$$

where A is the normalization factor and is given by $A = \left(\frac{m}{2\pi a}\right)^{N/2}$ and we denoted $x(t_j) = x_j$ for simplicity.

As suggested in [2], we used the Monte Carlo integrator **vegas** to evaluate the propagator, as illustrated in the following methods (see also Appendix 4.1):

Listing 2.1: Implementation of the path integral using Monte Carlo integration

```
def integrand(self, x, x0=None, fixed=True):

        if fixed:
                S_lat = (self.m / (2 * self.a)) * (x[0] - x0) **
                    2 + self.a * self.V(x0)
                S_lat += (self.m / (2 * self.a)) * (x0 - x[-1])
                    ** 2 + self.a * self.V(x[-1])
        else:
                S_lat = (self.m / (2 * self.a)) * (x[0] - x[-1])
                    ** 2 + self.a * self.V(x[-1])

        S_lat += (self.m / (2 * self.a)) * np.sum((x[1:] - x
            [:-1]) ** 2) + self.a * np.sum(self.V(x[:-1]))

        A = (self.m / (2 * np.pi * self.a)) ** (self.N / 2)

        return A * np.exp(-1.0 * S_lat)

def vegas_integrate(self, x0=None, fixed=True, x_min = None,
    x_max = None):

        if x_min is None and x_max is None:
                x_min, x_max = -5, 5

        if x_min > x_max:
```

[1]Euclidean refers to the signature of the metric obtained by analytically continuing the time parameter to a purely imaginary value. This continuation, known as a Wick Rotation, transforms oscillatory terms into exponentially decaying terms, improving the convergence properties of the integral.

```
                raise ValueError("x_min cannot be greater than 
                    x_max.")

        if fixed and x0 is None:
                raise ValueError("x0 must be provided when fixed=
                    True")

        if fixed:
                lims = [[x_min, x_max]] * (self.N - 1)
                func = partial(self.integrand, x0=x0, fixed=True)
        else:
                lims = [[x_min, x_max]] * self.N
                func = partial(self.integrand, fixed=False)

        integ = vegas.Integrator(lims)
        S_lat = integ(func, nitn=10, neval=100000)

        return S_lat.mean
```

The propagator is evaluated in the range $x \in [-5, 5]$ to keep the computational time reasonable, while ensuring that the result is not significantly affected, as suggested by the author. The integrator is run for 10 iterations and 10^5 Monte Carlo evaluations, with $a = 0.5$ and $N = 8$. This procedure is applied to both the harmonic and quartic potentials, as shown in Fig. 2.1a. A smaller value of a is then implemented to achieve a better approximation. The accuracy of the results obtained with the `vegas()` algorithm improves when increasing the number of iterations (`nitn`) and Monte Carlo evaluations (`neval`) because these parameters directly affect the precision of the adaptive integration. Setting $a = 0.4$ and $N = 10$ to keep the total "time" $T = a \cdot N = 4$ uneffected, one obtains a smaller and more oscillatory integrand due to the exponential suppression in the path integral. This makes it harder for the algorithm to efficiently sample the most relevant regions of the integration domain without sufficient evaluations. By increasing `nitn` and `neval` to 15 and 5×10^5 respectively, `vegas()` can better refine the weight distribution and reduce fluctuations, yielding more stable and accurate results, as shown in Fig. 2.1b

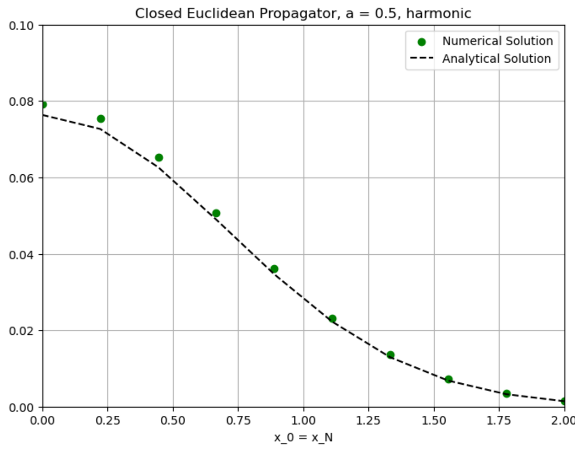
The need for the boolean variable `fixed` arises from the requirement of comparing numerical results with analytical ones. We cross-checked our implementation by adding a method to calculate the value of the ground state energy, which is expected to be $E_0 = 1/2$ when $\omega = 1$. To do so, we used the fact that

$$\int dx \, \langle x|e^{-\hat{H}T}|x\rangle \quad \underset{T\to\infty}{\longrightarrow} \quad e^{-E_0 T}, \tag{2.3}$$

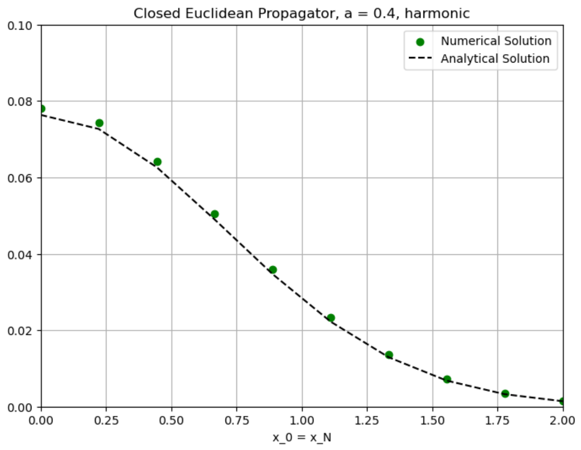
and that $T = aN$ to rewrite:

$$e^{-TE_0} = e^{-aN/2},$$

where we set $E_0 = 1/2$. The code for this procedure is shown here:



(a)



(b)

Figure 2.1: Harmonic oscillator propagator for: (a) $a = 0.5$ and $N = 8$, (b) $a = 0.4$ and $N = 10$.

Listing 2.2: Ground State energy evaluation

```
def analytic(self, x):
        x = np.array(x)
        return (np.exp(-1.0 * x ** 2) / np.sqrt(np.pi)) * np.exp
            (-1.0 * 1 / 2 * self.a * self.N)

def compute_analysis(self, analytical=True):
        expTE = self.vegas_integrate(fixed=False)
        E0 = -1.0 * np.log(expTE) / (self.a * self.N)
```

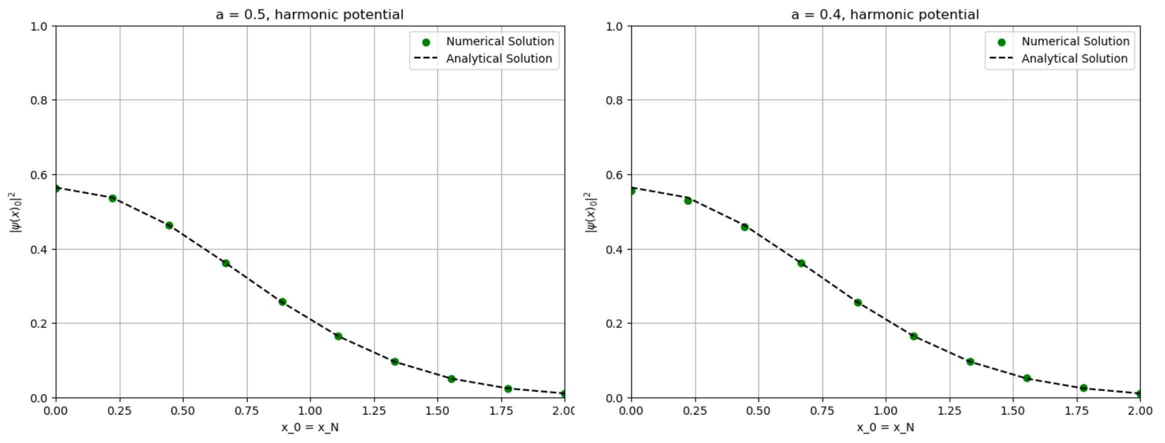
The boolean variable `fixed` allows changing `vegas_integrate()` to a method where no boundary conditions must be imposed (`fixed = False`). This method will then evaluate the integral for all periodic boundary conditions, ensuring the correct expression for e^{-TE_0}. `fixed = True` is used when boundary conditions must be chosen from a given array, as suggested for the propagator. The result is in perfect agreement with the expected value, as the ground state energy obtained by the simulation reported in Fig. 2.1 yields $E_0 = 0.4921$.

Lastly, we compared the ground state wave function that we obtained with the one derived analytically for the harmonic oscillator, i.e.:

$$\langle x|n\rangle = \frac{2^n n!}{\pi^{1/4}} e^{-x^2/2} H_n(x) \quad \rightarrow \quad \langle x|0\rangle = \frac{1}{\pi^{1/4}} e^{-x^2/2}. \tag{2.4}$$

The numerical results are again in perfect agreement with the theoretical predictions (Fig. 2.2), which confirms that the methods are well implemented.

The results for the quartic potential case, i. e. $V(x) = x^4/2$ are shown in Fig. 2.3



(a) (b)

Figure 2.2: Harmonic oscillator ground state wavefunction for: (a) $a = 0.5$ and $N = 8$, (b) $a = 0.4$ and $N = 10$. Analytical results are obtained by exploiting (2.4)

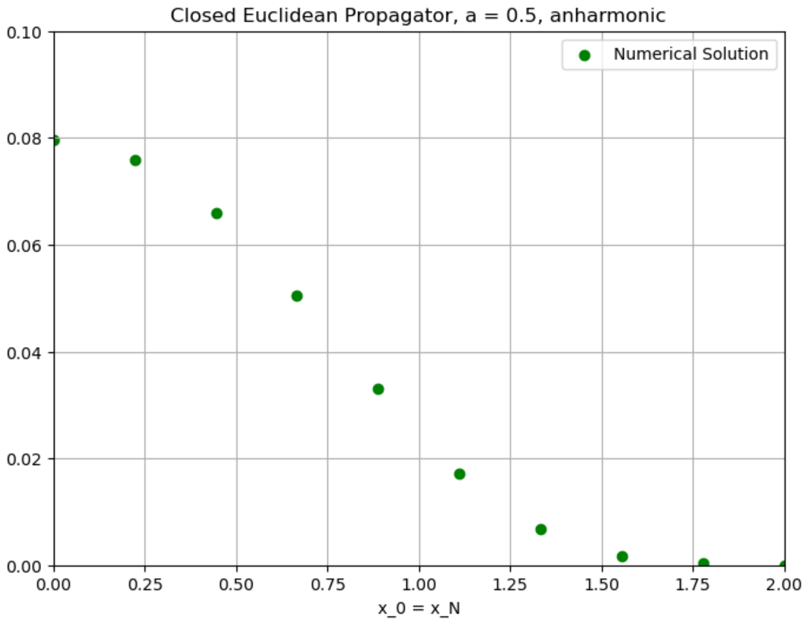


Figure 2.3: Numerical results for the quartic potential propagator for $a = 0.5$ and $N = 8$

2.2 Monte Carlo Evaluation of Path Integrals

In order to address and solve the second problem highlighted at the beginning of this chapter, one typically employs Monte Carlo sampling. This is a widely used computational method where calculations are performed based on a randomized selection from a given distribution. In the current context, it will be used to evaluate the first excited state of the problem presented above, as the ground state in Quantum Field Theory (QFT) corresponds to the vacuum and is therefore of limited interest.

To achieve this, one introduces the n-point correlation function, which is used to describe several physical observables. It is generally defined as:

$$\langle x(t_1)\dots x(t_n)\rangle = \frac{1}{Z[0]}\int \mathcal{D}x\, x(t_1)\dots x(t_n)e^{-S_E[x]}, \quad \text{with} \quad Z[0]=\int \mathcal{D}x\, e^{-S_E[x]}. \tag{2.5}$$

Of particular interest is the 2-point correlation function, also called the propagator, whose numerator can be expressed, referring to canonical quantization, as:

$$\int dx\, \langle x|e^{-\hat{H}(t_f-t_2)}\tilde{x}e^{-\hat{H}(t_2-t_1)}\tilde{x}e^{-\hat{H}(t_1-t_i)}|x\rangle\,, \tag{2.6}$$

where we integrate over all $x_i = x_f = x$ as well as the intermediate $x(t)$ values. By setting $T \equiv t_f - t_i$ and $t = t_2 - t_1$, the propagator can be discretized as follows:

$$\langle x(t_2)x(t_1)\rangle = \frac{\sum e^{-E_nT}\,\langle E_n|\tilde{x}e^{-(\hat{H}-E_n)t}\tilde{x}|E_n\rangle}{\sum e^{-E_nT}}. \tag{2.7}$$

If we now assume that both t and T are sufficiently large, but with $t \ll T$, one obtains [2]:

$$G(t) \equiv \langle\langle x(t_2)x(t_1)\rangle\rangle \underset{t\,\text{large}}{\longrightarrow} |\langle E_0|\tilde{x}|E_1\rangle|^2 e^{-(E_1-E_0)t}, \tag{2.8}$$

from which the first excitation energy can be extracted from the large-t dependence of $G(t)$:

$$\Delta E \equiv E_1 - E_0 = \frac{1}{a}\log\left(\frac{G(t)}{G(t+a)}\right), \tag{2.9}$$

allowing one to also determine the transition matrix element $\langle E_0|\tilde{x}|E_1\rangle$. Two important remarks are in order:

- Path integral averages of any functional $\Gamma[x]$ can be used to compute physical properties of excited states.
- By interpreting T as $\beta = \frac{1}{k_BT_{\text{temp}}}$, it becomes evident that any computer code designed for path integrals can also be applied to thermal physics problems.

Now let's take a closer look at the second code we implemented. The aim of the code is to implement the Metropolis Monte Carlo algorithm for a one-dimensional harmonic oscillator and use it to evaluate the propagator, in the form:

$$G(t) = \sum_j x(t_j+t)x(t_j), \tag{2.10}$$

Our implementation of the algorithm 4.2, strongly influenced by the author's work, is as follows[2]

Listing 2.3: Metropolis Algorithm

```
def update(self):
        accepted_moves = 0  # Counter for accepted moves
        total_moves = 0     # Counter for total moves

        for j in range(self.N):
                old_x = self.x[j]
                old_Sj = self.S(j, self.x)
                self.x[j] += np.random.uniform(-self.eps, self.
                    eps)  # Propose a new value for x[j]
                dS = self.S(j, self.x) - old_Sj  # Compute the
                    change in the action
                total_moves += 1  # Increment the total moves
                    counter
                if dS <= 0 or np.exp(-dS) >= np.random.uniform(0,
                    1):
                        accepted_moves += 1  # Increment the
                            accepted moves counter if accepted
                else:
                        self.x[j] = old_x  # Revert to the
                            previous value if not accepted

        acceptance_rate = accepted_moves / total_moves if
            total_moves > 0 else 0

        return acceptance_rate
```

The idea behind the algorithm is to create a large number N_{cf} of random paths (configurations) $\{x^{(\alpha)}\}$ with probability:

$$P[x^{(\alpha)}] \propto e^{-S[x^{(\alpha)}]}. \tag{2.11}$$

The Metropolis Algorithm allows achieving this by starting with a single configuration. By choosing an arbitrary path $x^{(0)}$, we can generate a new random path by randomizing the x_j's at each site of the lattice, in the way described in [2][3] or as shown in the code above. Once all the sites have been randomized, a new path is created, and the process is repeated N_{cf} times.
This algorithm must be used carefully. Indeed, the value of the parameter `eps` (ϵ) is of fundamental importance to obtain the desired results. For large values of ϵ, the changes will usually be significant and, therefore, often rejected. On the contrary, if ϵ is too small, the changes will often be accepted, but the new values at each x_j will be

[2]All comments of the various methods have been removed when transcribed here; they can be found in the GitHub repository, in case further clarifications are needed.

[3]Page 7

almost identical to the old ones. As suggested by LePage, we chose a value for ϵ such that $48\% - 50\%$ of the x_j values are modified. This range of values is calculated[4] using the `acceptance_rate` variable, which is averaged over the number of iterations in the `Mcaverage()` method.
To reduce the statistical correlation between successive paths, only those separated by `N_cor` steps are stored. Different values of this parameter are tested until the standard deviation becomes almost independent of it. In our case, we found that `N_cor = 20` is a suitable value to achieve this result and is therefore the one used to obtain Fig. 2.4.

2.2.1 Statistical Errors

There are several possible approaches to extract the most significant statistical data. In principle, one could, for instance, calculate an estimate of the mean of any functional $\Gamma[x]$ provided by the Monte Carlo method through:

$$\langle\langle\Gamma[x]\rangle\rangle \approx \bar{\Gamma} \equiv \frac{1}{N_{\text{cf}}} \sum_{\alpha=1}^{N_{\text{cf}}} \Gamma[x^{(\alpha)}]. \tag{2.12}$$

Such an estimate is, however, never exact. There are several statistical errors that would be canceled out only in the case of infinite configurations ($N_{\text{cf}} \to \infty$). The analysis of these statistical errors is an integral part of a functioning Monte Carlo method and, therefore, cannot be neglected. To avoid procedures that require excessive compilation times, such as repeating the calculation of the average over a sufficiently large number of randomly generated ensembles, one can use the methods presented below:

Listing 2.4: Function to reduce correlation between different paths

```
def bin(self, G, binsize):
        return np.array([np.mean(G[i:i + binsize], axis=0) for i
            in range(0, len(G), binsize)])

def bootstrap(self, G):
        N_bs = len(G)
        return G[np.random.randint(0, N_bs, size=N_bs)]

def bootstrap_deltaE(self, G, nbstrap=100):
        bsE = np.empty((nbstrap, self.N - 1))
        for i in range(nbstrap):
                g = self.bootstrap(G)
                bsE[i] = self.deltaE(self.avg(g))
        return self.avg(bsE), self.sdev(bsE)
```

[4]We did not calculate the exact boundaries; we simply observed during the simulations that the ϵ value always fell within this range.

The bootstrap method is a resampling technique that allows us to estimate the uncertainties in the calculated quantities, such as the averaged correlation function $G(n)$ and the energy differences ΔE. In the context of our code, the bootstrap generates multiple resampled ensembles of the correlation functions G by randomly selecting configurations (with replacement) from the original ensemble. [5]. This process is repeated a large number of times (e.g., $N_{\text{bstrap}} = 1000$), and for each resampled ensemble, we compute the relevant quantities. Finally, the standard deviation of the values obtained from the resampled ensembles provides an estimate of the uncertainty. The use of the bootstrap is essential because it helps account for residual correlations between configurations and provides a robust estimation of statistical errors, even in cases where analytical error propagation may be challenging.
This method is complemented by the binning procedure, which is excellent for removing (or reducing to a very low value) the correlation between different configurations. Instead of storing $G^{(1)}, G^{(2)}, G^{(3)}$... we could store:

$$\bar{G}^{(1)} \equiv \frac{G^{(1)} + G^{(2)} + G^{(3)} + G^{(4)}}{4} \tag{2.13}$$

$$\bar{G}^{(2)} \equiv \frac{G^{(5)} + G^{(6)} + G^{(7)} + G^{(8)}}{4} \quad \dots \tag{2.14}$$

This way, the statistical properties remain unchanged while the new set is far less numerous than the original one. This also ensures that, if the Monte Carlo estimates are binned with sufficiently large bins, the majority of estimates in one bin will be uncorrelated from the majority in adjacent bins.
Finally, another issue must be addressed. Starting the simulation is not straightforward, as the initial configuration is often atypical and could bias the results of the entire simulation. To mitigate this, a number of initial configurations are discarded in a process known as *thermalizing the lattice*, which is implemented within the `MCaverage()` method

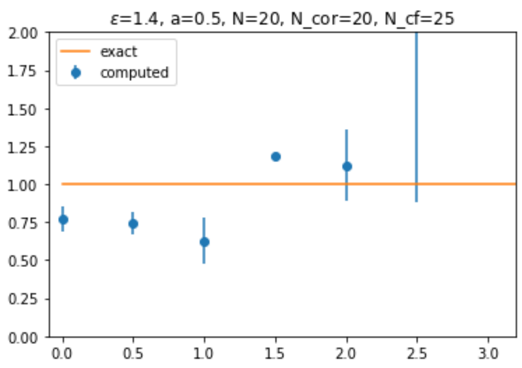
Listing 2.5: Monte Carlo Averages

```
def MCaverage(self, x, G):
        x.fill(0)
        total_acceptance_rate = 0  # Accumulator for acceptance
            rates
        num_updates = 0  # Counter for the number of updates

        # Thermalization step
        for _ in range(10 * self.N_cor):
                total_acceptance_rate += self.update()
                num_updates += 1

        # Compute configurations
```

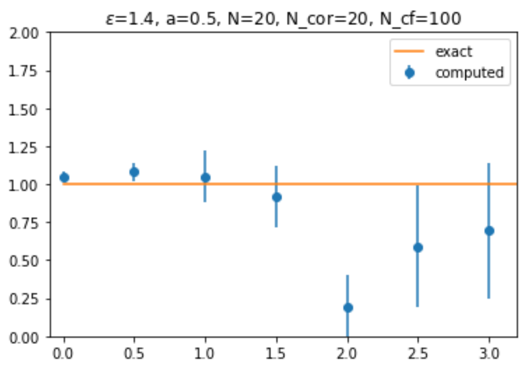
[5]It is important to note that, by doing so, some configurations $G^{(\alpha)}$ may appear multiple times in an ensemble, while others may not appear at all.

```
for alpha in range(self.N_cf):
        for _ in range(self.N_cor):
                total_acceptance_rate += self.update()
                num_updates += 1
        for n in range(self.N):
                G[alpha][n] = self.compute_G(x, n)

# Calculate the average acceptance rate
avg_acceptance_rate = total_acceptance_rate / num_updates

# Print the average acceptance rate
print(f"Average acceptance rate over all updates: {
    avg_acceptance_rate * 100:.2f}%")
```

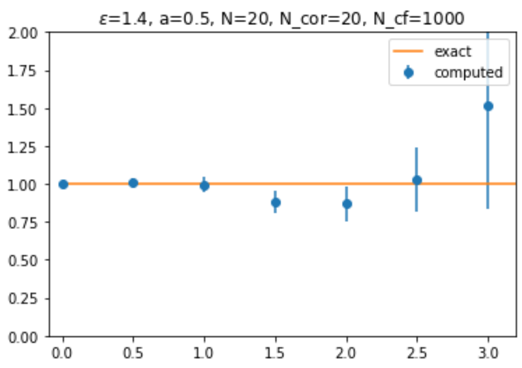
The results are the following:



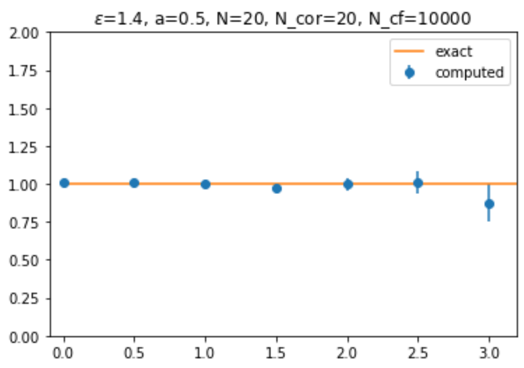
(a)



(b)



(c)



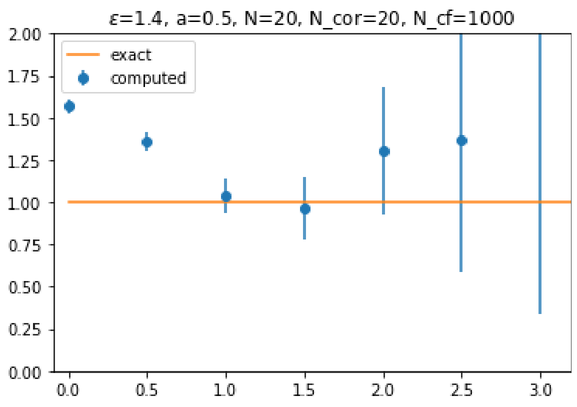
(d)

Figure 2.4: ΔE results for different values of N_{cf}.

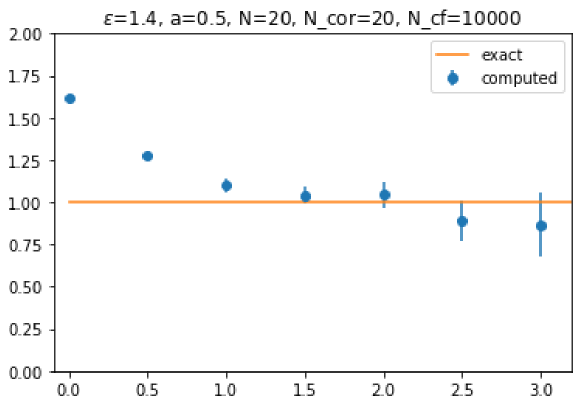
As expected, higher values of N_{cf} guarantee a more accurate sampling of the path integral and, therefore, better results.

2.2.2 $x^3(t)$ Case

The following exercise required repeating the analysis but with x^3, rather than x, as the source and the sink. The results are shown in Fig. 2.5.
The main difference with the previous case is that $\Delta E(t)$ converges to the same result



(a)



(b)

Figure 2.5: ΔE results for different values of N_{cf} when x^3 is used as the source and sink.

as before, but for larger values of t. Therefore, only the $N_{\text{cf}} = 10^3$ and $N_{\text{cf}} = 10^4$ cases are reported, as smaller values led to almost random results.

2.3 Different Discretization Methods

By referring to [2] for a much deeper discussion on the subject, we now address the question of how to minimize the magnitude of discretization errors and how different discretization schemes influence the results of simulations. In a lattice approximation, the value of the field ϕ is given only at the lattice sites, and therefore derivatives must be replaced by finite differences. The first approximation presented in such cases is:

$$\left.\frac{\partial^2\phi(x)}{\partial x^2}\right|_{x=x_j} \approx \Delta_x\phi(x_j) + \mathcal{O}(a^2) = \left.\frac{\phi(x+a) - 2\phi(x) + \phi(x-a)}{a^2}\right|_{x=x_j} + \mathcal{O}(a^2). \quad (2.15)$$

If a more accurate approximation is required for lattice calculations, one typically uses the following discretization formula:

$$\left.\frac{\partial^2\phi(x)}{\partial x^2}\right|_{x=x_j} \approx \Delta_x\phi(x_j) - \frac{a^2}{12}(\Delta_x)^2\phi(x_j) + \mathcal{O}(a^4), \quad (2.16)$$

which generally provides sufficient precision for most scenarios. However, there is an important caveat to consider when using this improved discretized version. Suppose one wishes to derive the discretized Euclidean classical equation of motion from the action via:

$$\left.\frac{\partial S[x]}{\partial x}\right|_{x=x_j} = 0. \quad (2.17)$$

Using the explicit form of the improved action, the equations of motion for the harmonic oscillator become:

$$m(\Delta - \frac{a^2}{12}\Delta^2)x_j = \left.\frac{dV(x)}{dx}\right|_{x=x_j} = m\omega_0^2 x_j. \quad (2.18)$$

If one assumes a solution of the form $x_j = \exp(-\omega t_j) = \exp(-\omega a j)$ and applies the improved discretized derivative

$$(\Delta - \frac{a^2}{12}(\Delta)^2)x_j = \frac{x_{j+1} - 2x_j + x_{j-1}}{a^2} - \frac{1}{12}\frac{x_{j+2} - 4x_{j+1} + 6x_j - 4x_{j-1} + x_{j-2}}{a^2}, \quad (2.19)$$

the equations of motion can be rewritten as:

$$e^{-2\omega a} - 16e^{-\omega a} + 30 - 16e^{\omega a} + e^{2\omega a} = -12(a\omega_0)^2, \quad (2.20)$$

for the improved discretized action case. Unlike the unimproved formula, which admits a unique solution of the form:

$$\omega^2 \approx \omega_0^2\left(1 - \frac{1}{12}(a\omega_0)^2\right), \quad (2.21)$$

obtained via an iterative method, (2.20) admits two different solutions, only one of which is physical. The physical solution is derived as follows: starting with (2.20), one proceeds by writing:

$$\begin{aligned}
&\cosh(2\omega a) - 16\cosh(\omega a) + 15 = -6(a\omega_0)^2, \\
&\text{and using the expansion } \cosh(x) \approx 1 + \frac{x^2}{2} + \frac{x^4}{24} + \frac{x^6}{720} + \mathcal{O}(x^8), \\
&\omega^2 - \frac{1}{90}\omega^6 a^4 = \omega_0^2, \\
&\omega^2 \approx \omega_0^2 \left(1 + \frac{1}{90}(a\omega_0)^4\right). \qquad (2.22)
\end{aligned}$$

There are two key differences compared to the iterative solution for the unimproved action (2.21):

1. The errors are of fourth order in $a\omega_0$ instead of second order.
2. ω_0 now represents a lower bound for ω instead of an upper bound.

As anticipated, a second solution exists, corresponding to the zeros of the function:

$$f(\omega) = e^{-2\omega a} - 16e^{-\omega a} + 30 - 16e^{\omega a} + e^{2\omega a} \quad \rightarrow \quad \omega^2 \approx \left(\frac{2.6}{a}\right)^2. \qquad (2.23)$$

This solution corresponds to a new oscillation mode absent in the continuum, an artifact of the improved lattice theory sometimes referred to as a "numerical ghost." It arises from the discretization of temporal derivatives and represents a spurious state with negative norm, which lowers the ΔE values, making them unreliable as upper bounds.

To eliminate these extra states, a change of variables is performed in the path integral:

$$x_j = \tilde{x_j} + \delta\tilde{x_j} \equiv \tilde{x_j} + \xi_1 a^2 \Delta\tilde{x_j} + \xi_2 a^2 \omega_0^2 \tilde{x_j}, \qquad (2.24)$$

with a unitary Jacobian that does not introduce additional terms in the action. Assuming $\delta\tilde{x_j}$ is small, the action becomes:

$$\begin{aligned}
S[x] &= S[\tilde{x} + \delta\tilde{x}] \\
&= S[\tilde{x}] + \sum_j \frac{\partial S}{\partial \tilde{x_j}}\delta\tilde{x_j} + \mathcal{O}(a^4) \equiv \tilde{S}[\tilde{x}]. \qquad (2.25)
\end{aligned}$$

After simplifications and setting $\xi_1 = 0$ and $\xi_2 = 1/12$, the improved action becomes:

$$\tilde{S}_{\text{imp}}[\tilde{x}] = \sum_{j=0}^{N-1} a\left[\frac{1}{2}m\tilde{x_j}\Delta\tilde{x_j} + \tilde{V}_{\text{imp}}\right], \quad \text{with} \quad \tilde{V}_{\text{imp}} = \frac{1}{2}m\omega_0^2\tilde{x_j}^2\left(1 + \frac{(a\omega_0)^2}{12}\right). \quad (2.26)$$

This action eliminates a^2 errors and ghost states, restoring the expected behaviour in which the asymptotic value of ΔE is reached from above. Using this form of the action, the frequency ω of our ansatz is approximated by:

$$\omega^2 = \omega_0^2 \left(1 - \frac{(a\omega_0)^4}{360}\right). \tag{2.27}$$

2.3.1 Code Implementation

The `EOMSolver` Class [1] 4.3 is implemented in order to study numerically and derive all the results described in the previous paragraph, focusing on the harmonic oscillator case. Due to the linearity of the equations of motion, we opted for a matrix-based resolution of the problem, as shown below.

Listing 2.6: EOM calculations for the harmonic potential

```python
def build_matrix(self, a):

    matrix = np.zeros((self.N, self.N))  # Initialize an NxN matrix

    if self.mode == 'n':
        diagonal = -(2 + a**2 * self.w0**2)  # Main diagonal elements
        off_diagonal = 1  # Off-diagonal elements

        for i in range(self.N):
            matrix[i, i] = diagonal  # Main diagonal
            if i > 0:
                matrix[i, i - 1] = off_diagonal  # Lower diagonal
            if i < self.N - 1:
                matrix[i, i + 1] = off_diagonal  # Upper diagonal

    elif self.mode == 'y':
        coeff_j = -(5 / 2 + a**2 * self.w0**2)  # Main diagonal
        coeff_j_pm_1 = 4 / 3  # First off-diagonals
        coeff_j_pm_2 = -1 / 12  # Second off-diagonals

        for i in range(self.N):
            matrix[i, i] = coeff_j
        if i > 0:
            matrix[i, i - 1] = coeff_j_pm_1
        if i < self.N - 1:
            matrix[i, i + 1] = coeff_j_pm_1
        if i > 1:
            matrix[i, i - 2] = coeff_j_pm_2
```

```
                if i < self.N - 2:
                        matrix[i, i + 2] = coeff_j_pm_2

        elif self.mode == 'no_ghost':
                diagonal = -(2 + (a * self.w0)**2 * (1 + (a *
                    self.w0)**2 / 12))  # Main diagonal elements
                off_diagonal = 1  # Off-diagonal elements

                for i in range(self.N):
                        matrix[i, i] = diagonal  # Main diagonal
                if i > 0:
                        matrix[i, i - 1] = off_diagonal  # Lower
                            diagonal
                if i < self.N - 1:
                        matrix[i, i + 1] = off_diagonal  # Upper
                            diagonal

        return matrix
```

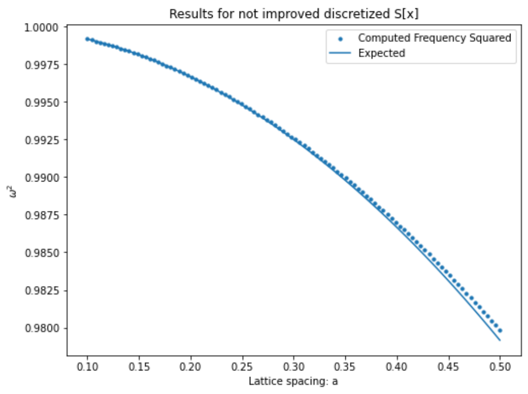
The `build_matrix` method generates a matrix used to discretize the problem, depending on the selected mode (`'n'`, `'y'`, or `'no_ghost'`). Each mode defines specific coefficients for the matrix to adapt the resolution method (e.g., tridiagonal or pentadiagonal) to the required accuracy or problem constraints. The resulting matrix is returned for further computation.
In order to solve any differential equation, boundary conditions must be imposed. In matrix form, this is fairly simple to achieve, as one needs to require that the product between the matrix and the vector of solutions simply equals the vector of boundary conditions.

Listing 2.7: EOM boundary conditions and solutions to the problem

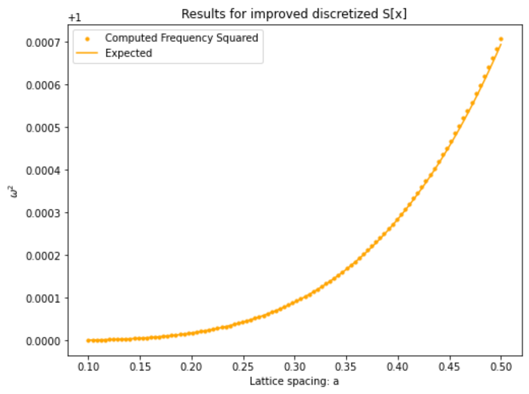
```
def boundary_conditions(self, xi, xf, xi2, xf2):

        B = np.zeros(self.N)

        if self.mode in ['n', 'no_ghost']:
                B[0] = -xi
                B[-1] = -xf

        elif self.mode == 'y':
                B[0] = (1 / 12) * xi2 - (4 / 3) * xi
                B[1] = (1 / 12) * xi
                B[-2] = (1 / 12) * xf
                B[-1] = (1 / 12) * xf2 - (4 / 3) * xf

        return B

        (...)

def perform_analysis(self):
```

```
w2 = np.empty(len(self.a))
for i, ai in enumerate(self.a):
        t = np.arange(0, self.N) * ai
        xi = np.exp(0)  # Boundary condition at t_0
        xf = np.exp(-self.T[i])  # Boundary condition at
            t_N
        xi2 = np.exp(-(-ai))  # Condition at t_{-2}
        xf2 = np.exp(-self.T[i] + ai)  # Condition at t_{
            N+1}

        A = self.build_matrix(ai)
        B = self.boundary_conditions(xi, xf, xi2, xf2)
        X = np.linalg.solve(A, B)

        w2[i] = self.fit_solution(X, t)

return w2
```

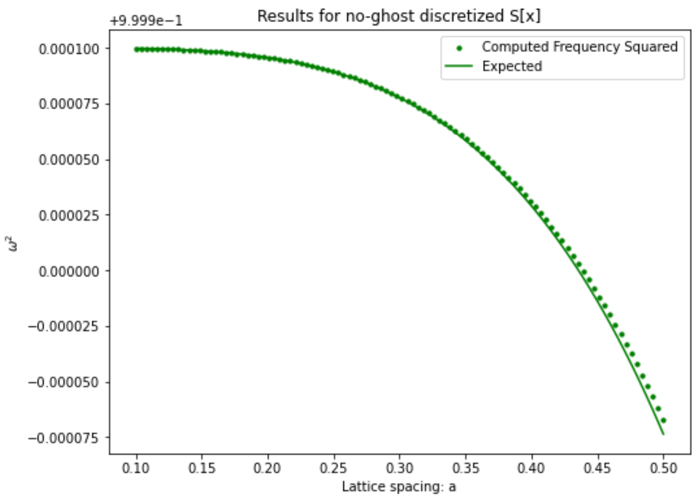
The comparison between the expected results (2.21), (2.22), (2.27) and the results obtained by our code are shown in Fig. 2.6.



(a) ω^2 behaviour as a function of the lattice spacing a for the unimproved action



(b) ω^2 behaviour as a function of the lattice spacing a for the improved action



(c) ω^2 behaviour as a function of the lattice spacing a for the transformed action

Figure 2.6: ω^2 behaviour as a function of the lattice spacing `a=np.linspace(0.1, 0.5, 100)` obtained via `EOMSolver` Class compared with theoretical results

A single plot with all three results is also included to provide a clear visual comparison of the different methods (Fig. 2.7)

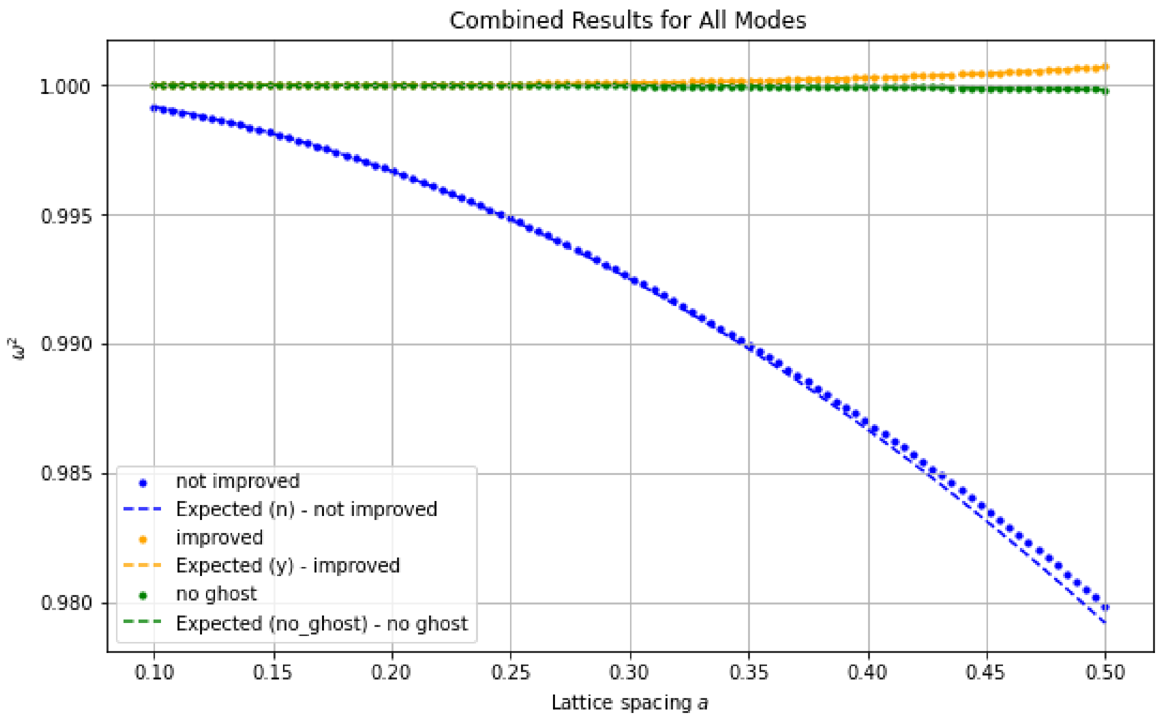


Figure 2.7: All three results shown in Fig. 2.6 combined in a unique graph

As can be seen from the previous plots, all numerical results are in perfect agreement with the theoretical ones. This demonstrates the accuracy and reliability of the implemented methods, confirming that the code correctly reproduces the expected behavior across all tested scenarios.

To further test the validity of the theoretical discussion presented in the first section of paragraph 2.3, we implemented the method `compute_ghost_mode()` to test the existence of the "numerical ghost", its behavior with respect to the lattice spacing, and to evaluate the value of the constant C in the solution of the form $\omega \approx C/a$. The method is reported below, and our results are shown in Fig. 2.8.

Listing 2.8: numerical ghost evaluation and behaviour

```
def compute_ghost_mode(self):

        ghost = np.empty(len(self.a))

        for i, ai in enumerate(self.a):
                def f(w):
                # Ghost mode equation from the improved
                    discretization
                        return (
                                np.exp(-2 * ai * w) - 16 * np.exp
                                    (-ai * w) + 30
                                - 16 * np.exp(ai * w) + np.exp(2
                                    * ai * w)
                                )

                # Use a better initial guess close to the
                    theoretical ghost mode
                initial_guess = 2.6 / ai

                # Solve numerically, using the improved guess
                sol = optimize.fsolve(f, initial_guess)

                ghost[i] = sol[0]  # Store the solution

        # Calculate w^2
        ghost_squared = ghost**2

        # Define the theoretical expectation for ghost mode
        def expectation(a, C):
                return (C / a)**2

        # Fit the numerical ghost modes to the theoretical form
        popt, _ = optimize.curve_fit(expectation, self.a,
            ghost_squared)
        C_fit = popt[0]  # Extract the fitted constant

        # Generate the fitted curve for w^2
        fit_squared = expectation(self.a, C_fit)

        (... functions for plotting ...)
```

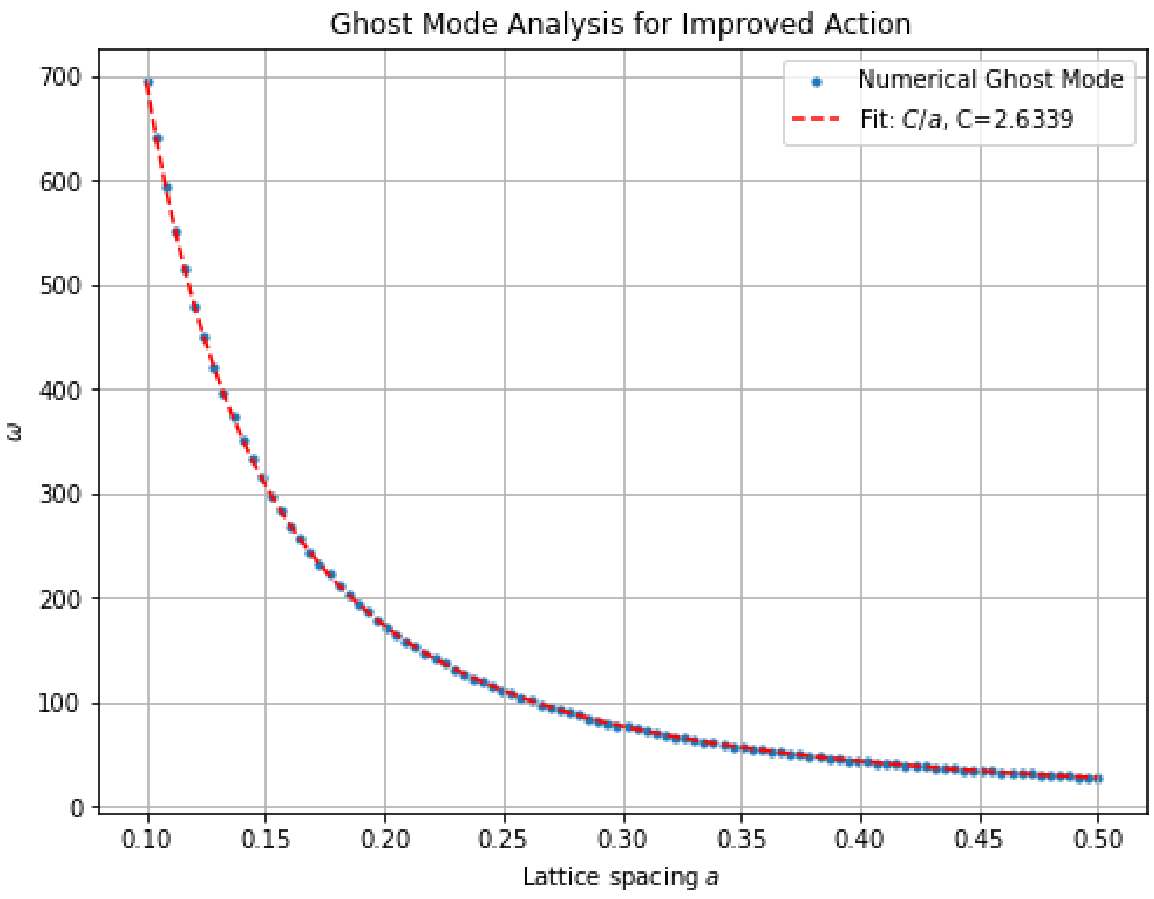


Figure 2.8: ω^2 ghost solution behaviour wrt the lattice spacing `a=np.linspace(0.1, 0.5, 100)`. The numerical value of the constant C is found to be ≈ 2.6339

In this case as well, the numerical results are in perfect agreement with the theoretical predictions: the value of the proportionality constant and the behavior of ω with respect to the lattice spacing are in perfect agreement with what discussed in [2].

2.3.2 Anharmonic Potential and Excitation Energy with improved methods

When trying to extend what was discussed at the beginning of paragraph 2.3 to the case of an anharmonic potential of the form

$$V_{\text{an}} = \frac{1}{2} m\omega_0^2 x^2 \left(1 + cm\omega_0 x^2\right), \tag{2.28}$$

where c is a dimensionless parameter, few complications arise. One can, in principle, apply the same exact procedure for the unimproved and improved action. What changes is how to solve the equations of motion. The latter will indeed have a x^3 dependence, which forbids the usage of a matrix-based type of solution and imposes the usage of numerical procedures to find the solutions for such equations. Moreover, when trying to eliminate numerical ghost states, one needs to perform a change of variables of the form:

$$\delta\tilde{x}_j \equiv \delta\tilde{x}_j^{\text{har}} + \xi_3 a^2 m\omega_0^3\tilde{x}_j^3 = \xi_1 a^2\Delta\tilde{x}_j + \xi_2 a^2\omega_0^2\tilde{x}_j + \xi_3 a^2 m\omega_0^3\tilde{x}_j^3, \tag{2.29}$$

which introduces a non-unitary Jacobian factor, resulting in extra terms in the form of the potential. We did not develop a full analysis of the equations of motion for the anharmonic potential; however, we did repeat the simulations for the excitation energy ΔE, based on the Monte Carlo method, for both the anharmonic and the harmonic potential. In order to do so, we modified our `PathIntegralMonteCarlo` class to take into account also the more advanced discretization methods:

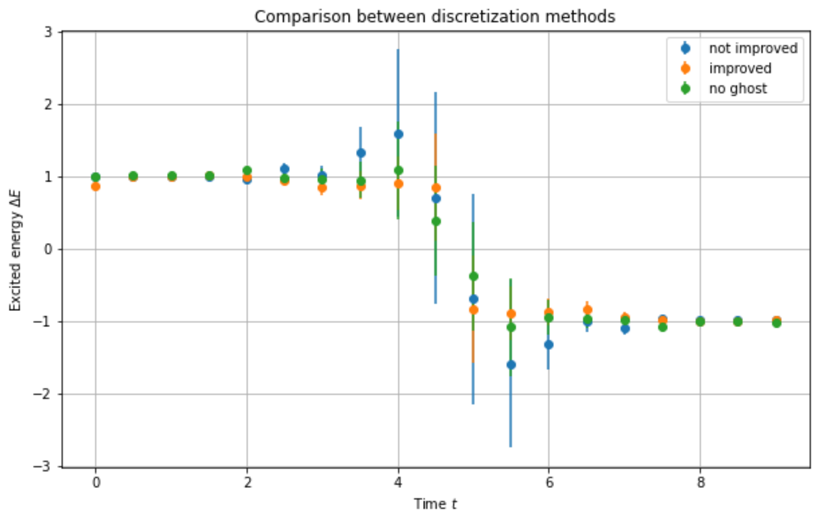
Listing 2.9: numerical ghost evaluation and behaviour

```
def S(self, j, x):

        jp = (j + 1) % self.N
        jm = (j - 1) % self.N
        jm2 = (j - 2) % self.N
        jp2 = (j + 2) % self.N

        if self.potential == 'harmonic':

                if self.imp == 'n':
                        return self.a * self.m * (self.w ** 2) *
                            x[j] ** 2 / 2 + self.m * x[j] * (x[j]
                            - x[jp] - x[jm]) / self.a
                elif self.imp == 'y':
                        return self.a * self.m * (self.w ** 2) *
                            x[j] ** 2 / 2 - (self.m / (2 * self.a)
                            ) * x[j] * (-(x[jm2] + x[jp2]) / 6 + (
                            x[jm] + x[jp]) * (8 / 3) - x[j] * (5 /
                             2))
                elif self.imp == 'noghost':
                        return self.a * self.m * (self.w ** 2) *
                            (1 + (self.a * self.w) ** 2 / 12) * x[
                            j] ** 2 / 2 + self.m * x[j] * (x[j] -
                            x[jp] - x[jm]) / self.a

        elif self.potential == 'anharmonic':
                c = 2  # anharmonicity constant

                if self.imp == 'n':
                        return 0.5 * self.a * self.m * (self.w **
                             2) * (x[j]**2) * (1 + c * self.m *
                            self.w * x[j]**2) + self.m * x[j] * (x
                            [j]-x[jp]-x[jm])/self.a
```

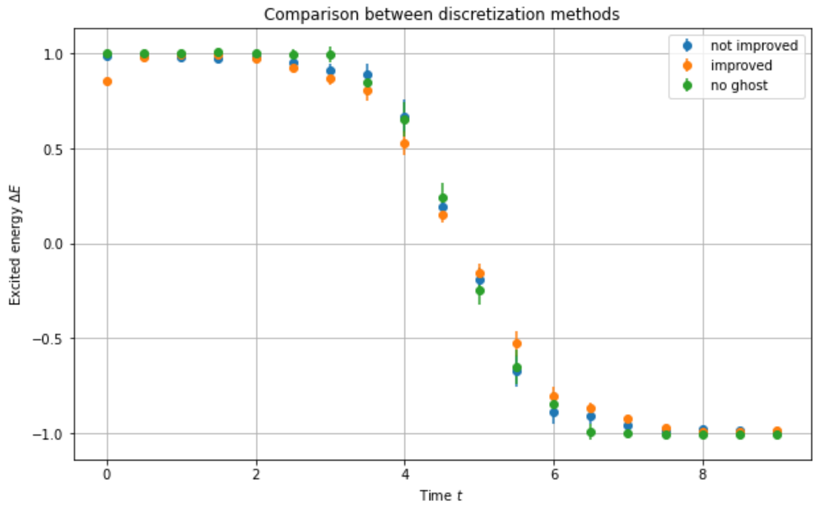
```
elif self.imp == 'y':
        S_kinetic = (
                - (self.m / (2 * self.a)) *
                x[j] * (
                - (x[jm2] + x[jp2]) / 6 +
                (8 / 3) * (x[jm] + x[jp]) -
                (5 / 2) * x[j]
                )
        )
        S_potential = (
                self.a * self.m * (self.w**2 / 2)
                    * x[j]**2 * (1 + c * self.m *
                    self.w * x[j]**2)
                )
        return S_potential + S_kinetic

elif self.imp == 'noghost':
        S_correction = (
                (self.a**2 / 24) * (x[j] + 2 * c
                    * self.m * self.w * x[j]**3)
                    **2  # Correzione per i modi
                    ghost
                - (self.a**2 / 4) * c * self.m *
                    self.w**3 * x[j]**2
                                 # Termine
                    correttivo quadratico
                + (self.a**3 / 2) * (0.25 * c *
                    self.m * self.w**3 * x[j]**2)
                    **2  # Termine correttivo
                    quartico
                )
        S_kinetic = self.m * x[j] * (x[j] - x[jp]
            - x[jm]) / self.a          # Termine
            cinetico
        S_potential = (
                self.a * self.m * (self.w**2 / 2)
                    * (1 + c * self.m * self.w**2
                    * x[j]**2) * x[j]**2  #
                    Potenziale migliorato
                )
        return S_potential + S_correction +
            S_kinetic
```

The results are presented below (Fig. 2.9 and Fig. 2.10). [6]

[6]Compare with results presented in Fig. 2.4, which are obtained by setting `imp = 'n'`.



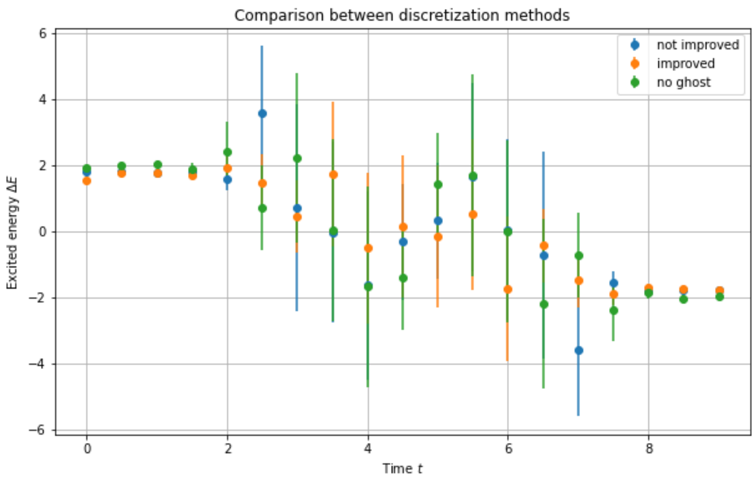
(a) `N_cf = `10^4



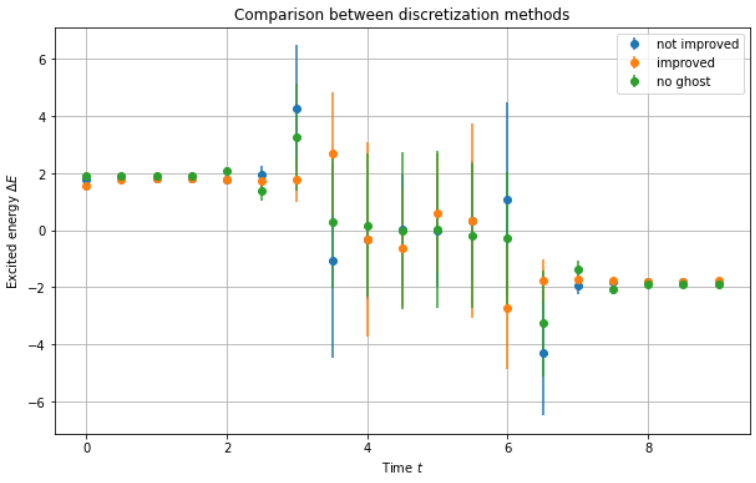
(b) `N_cf = `10^5

Figure 2.9: Comparison between ΔE results for different discretization methods for the harmonic potential. The values of the parameter are: $N = 20$, $\alpha = 0.5$, $\omega_0 = 1$, $\epsilon = 1.4$ and $N_{\text{cor}} = 20$ for both the simulations. Acceptance rate $\in [45, 50]\%$ for all methods.

As we've already pointed out, the ΔE values obtained via the improved method tend to reach the asymptotic value $\Delta E = 1$ from below, due to the presence of ghost states with negative norm.
We now present the results obtained for the anharmonic potential, which conclude the section about discretization methods.



(a) `N_cf = ` 10^4



(b) `N_cf = ` 10^5

Figure 2.10: Comparison between ΔE results for different discretization methods for the anharmonic potential. The values of the parameter are: $N = 20$, $\alpha = 0.5$, $\omega_0 = 1$, $\epsilon = 1.4$ and $N_{\text{cor}} = 20$ for both the simulations. Acceptance rate $\in [42, 46]\%$ for all methods.

Chapter 3

Field Theory On a Lattice

3.1 Classical and Quantum Gluons

Following the work of [2], we now attempt to apply the previously described methods for the 1D harmonic oscillator to a (3+1)D field theory, thus entering the realm of Lattice Quantum Chromodynamics (LQCD). In 1974, Wilson formulated Euclidean gauge theories on the lattice as a means to study confinement and to carry out non-perturbative analyses of QCD. The numerical implementation of the path integral approach requires the following five steps [4] [1]:

- Discretization of space-time.
- Transcription of the gauge and fermion degrees of freedom.
- Construction of the action.
- Definition of the integration measure in the path integral.
- Transcription of the operators used to probe physical observables.

Among these, the construction of the action and the operators is the most intricate step. The QCD action in Euclidean space-time is given by

$$S = \int d^4x \frac{1}{2} \sum_{\mu,\nu} \mathrm{Tr}\{G_{\mu\nu}^2\}, \quad \text{with} \quad G_{\mu\nu} = \partial_\mu b_\nu - \partial_\nu b_\mu + ig[b_\mu, b_\nu]. \tag{3.1}$$

Proving this relations requires few steps.
Wilson observed that, in the continuum, a fermion moving from site x to y in the presence of a gauge field picks up a phase factor given by

$$\psi(y) = \mathcal{P} e^{\int_x^y igb_\mu(x)dx_\mu} \psi(x), \tag{3.2}$$

where $\mathcal{P}$ denotes the path-ordered product. Equation (3.2) suggests that gauge fields should be associated with the links connecting sites on the lattice rather than being

[1]For the theoretical developement of this chapter we used this paper to integrate LePage's work.

defined at the sites themselves. However, this notation is somewhat imprecise, as it is impossible to formulate a lattice version of QCD directly in terms of $b_\mu(x)$ while maintaining exact gauge invariance. Consequently, we reformulate the theory in terms of $SU(3)$ matrices, which represent a discrete version of the path-ordered product:

$$U(x, x+\mu) \equiv U_\mu(x) = e^{iagb_\mu(x+\frac{\mu}{2})}. \tag{3.3}$$

The link variables $U_\mu(x)$ transform simply under a gauge transformation:

$$U_\mu(x) \to \Omega(x) U_\mu(x) \Omega^\dagger(x + a\mu), \tag{3.4}$$

and satisfy the relation

$$U(x, x-\mu) \equiv U_{-\mu}(x) = U^\dagger(x-\mu, x). \tag{3.5}$$

A link variable $U_\mu(x)$ is pictorially represented by a directed line from x to $x+\mu$, where this line corresponds to the integration path in the exponent of $U_\mu(x)$. In the conjugated matrix, the line integral is reversed, and thus the corresponding line should be considered as pointing in the opposite direction.

According to the definition in Eq. (3.3), an important gauge-invariant quantity that can be constructed on the lattice is the **Wilson loop**. A Wilson Loop, in Quantum Field Theory, is a gauge invariant operator arising from the parallel transport of gauge variables around closed loops. They encode all gauge information of the theory, allowing for the construction of loop representations which fully describe gauge theories in terms of these loop. The simplest example is the **plaquette**, a 1×1 loop:

$$W_{\mu\nu}^{1\times 1} \equiv P_{\mu\nu} = \Re\{\mathrm{Tr}\{U_\mu(x) U_\nu(x+\mu) U_\mu^\dagger(x+\nu) U_\nu^\dagger(x)\}\}. \tag{3.6}$$

For $SU(N \geq 3)$, the trace of any Wilson loop in the fundamental representation is generally complex, with the two possible path orderings yielding complex conjugate values. Taking the trace ensures gauge invariance, while taking the real part corresponds to averaging the loop and its charge-conjugate counterpart.
The reason why maintaining gauge invariance is fundamental, whereas one can easily forgo Lorentz invariance and other symmetries, is thoroughly discussed in [2] and will not be repeated here.
The significance of Eq. (3.6) becomes evident from the fact that the gauge action can be expressed in terms of closed loops. Considering, for simplicity, $U(1)$ matrices and substituting Eq. (3.3) into Eq. (3.6), one finds—after a Taylor expansion—that

$$\Re\{\mathrm{Tr}(1 - P_{\mu\nu})\} = \frac{a^4 g^2}{2} G_{\mu\nu} G^{\mu\nu}. \tag{3.7}$$

From this, it follows that

$$\frac{1}{g^2} \sum_x \sum_{\mu,\nu} \Re\{\mathrm{Tr}(1 - P_{\mu\nu})\} = \frac{a^4}{2} \sum_x \sum_{\mu,\nu} G_{\mu\nu} G^{\mu\nu} \to \frac{1}{4} \int d^4x G_{\mu\nu} G^{\mu\nu}. \tag{3.8}$$

The result for the gauge action in the $SU(3)$ case is given by

$$S_{\text{Wil}} = \frac{6}{g^2}\sum_x \sum_{\mu<\nu} \Re\left\{\text{Tr}\left(\frac{1}{3}(1-P_{\mu\nu})\right)\right\}, \tag{3.9}$$

from which it immediately follows that

$$W^{1\times 1}_{\mu\nu} = \frac{1}{3}\Re\left\{\text{Tr}\,\mathcal{P}e^{\oint_\Box -igb_\mu(x)dx_\mu}\right\} \approx \frac{1}{3}\Re\left\{\text{Tr}\left[1-\oint_\Box igb_\mu(x)dx_\mu - \frac{1}{2}\left(\oint_\Box igb_\mu(x)dx_\mu\right)^2 + \mathcal{O}(b^3)\right]\right\}. \tag{3.10}$$

We can further simplify this expression by applying Stokes' theorem:

$$\oint_\Box b_\mu(x)dx_\mu = \int_{-a/2}^{a/2} dx_\mu dx_\nu \left[\partial_\mu b_\nu(x+x_0) - \partial_\nu b_\mu(x+x_0)\right] \tag{3.11}$$

$$= \int_{-a/2}^{a/2} dx_\mu dx_\nu \left[G_{\mu\nu}(x_0) + (x_\mu D_\nu + x_\nu D_\mu)G_{\mu\nu}(x_0) + \dots\right] \tag{3.12}$$

$$= a^2 G_{\mu\nu}(x_0) + \frac{a^4}{24}(D_\mu^2 + D_\nu^2)G_{\mu\nu}(x_0) + \mathcal{O}(a^6, b^2), \tag{3.13}$$

where x_0 is an arbitrary point around which the expansion is performed, and D_μ is the gauge-covariant derivative. There are no G^3 terms since $P_{\mu\nu}$ is invariant under the transformation $U_\mu \to U_\mu^\dagger$.

By substituting Eq. (3.11) into Eq. (3.9), we obtain the formula:

$$S_{\text{Wil}} = \int d^4x \sum_{\mu<\nu}\left\{\frac{1}{2}\text{Tr}G_{\mu\nu}^2 + \frac{a^2}{24}\text{Tr}\left[G_{\mu\nu}(D_\mu^2 + D_\nu^2)G_{\mu\nu}\right] + \dots\right\}, \tag{3.14}$$

which correctly reproduces the continuum limit up to corrections of order a^2.

We can reduce the a^2 error by incorporating additional Wilson loops. For example, the $2a \times a$ loop, known as the **rectangle operator**, has the expansion:

$$W^{2\times 1}_{\mu\nu} \equiv R_{\mu\nu} = 1 - \frac{2}{3}a^4\text{Tr}(gG_{\mu\nu})^2 - \frac{1}{18}a^6\text{Tr}\left[gG_{\mu\nu}(4D_\mu^2 + D_\nu^2)gG_{\mu\nu}\right] - \dots \tag{3.15}$$

which allows us to construct an improved version of the classical action, reducing errors to order $o(a^4)$:

$$S \equiv -\beta\sum_x\sum_{\mu,\nu}\left\{\frac{5P_{\mu\nu}}{3} - \frac{R_{\mu\nu}+R_{\nu\mu}}{12}\right\} + \text{const} \tag{3.16}$$

$$= \int d^4x \sum_{\mu<\nu}\left\{\frac{1}{2}\text{Tr}G_{\mu\nu}^2\right\} + \mathcal{O}(a^4), \tag{3.17}$$

, with beta being defined as $\beta \equiv 6/g^2$. This completes the proof, as the result obtained is exactly (3.1).
When transitioning from classical to quantum analysis, the situation changes radically. Since gluonic operators are built from link operators rather than vector potentials, new unphysical vertices may arise. For example, the leading term in the Lagrangian that couples quarks and gluons is

$$\frac{\bar{\psi} U_\mu \gamma_\mu \psi}{a}, \tag{3.18}$$

which contains vertices with any number of additional powers of agb_μ. Such vertices are problematic since pairs of b_μ, if contracted with each other, give rise to ultraviolet-divergent terms. These are known as **tadpole contributions**. Such contributions spoil perturbation theory and, consequently, the connection between the continuum and the lattice. Therefore, one must remove them. This is achieved through **tadpole improvement**, which consists of properly redefining the gauge variables by normalizing the gauge links with the expectation value of the smallest possible Wilson loop, typically the average plaquette u_0:

$$u_0 = \langle 0 | \frac{1}{3} \mathrm{Tr} U_\mu | 0 \rangle^{1/4}. \tag{3.19}$$

Applying this redefinition to Eq. (3.16) results in the improved action:

$$S \equiv -\beta \sum_x \sum_{\mu,\nu} \left\{ \frac{5}{3} \frac{P_{\mu\nu}}{u_0^4} - \frac{1}{12} \frac{R_{\mu\nu} + R_{\nu\mu}}{u_0^6} \right\}. \tag{3.20}$$

When defined in this manner, the factors u_0 are gauge dependent. However, a numerically inexpensive and gauge-independent method exists, which relies on the following alternative definition:

$$u_0 = \langle 0 | P_{\mu\nu} | 0 \rangle^{1/4}. \tag{3.21}$$

Tadpole improvement is the first step in a systematic procedure for improving the action. The next step involves incorporating renormalization effects due to contributions from high-momentum modes ($k > \pi/a$) that are not already accounted for by tadpole improvement. These effects are mitigated by renormalizing the coefficient of the rectangle operator $R_{\mu\nu}$ in the action and by adding an additional operator, which can be chosen in several ways [2].

3.1.1 Code Implementation: Monte Carlo Evaluation of Gluonic Path Integrals

This paragraph refers to the implementation and logical structure of `WilsonLatticeUtils` Class 4.4.

In analogy with the approach used for the one-dimensional path integral, we now address the problem of performing a Monte Carlo evaluation of gluonic path integrals. The fundamental variables in this context are the link variables $U_\mu(x)$ (Eq. (3.3)), which must be updated using a randomly generated $SU(3)$ matrix M. These matrices are generated once at the beginning of the simulation by the function `generate_matrices_pool()`.

To ensure that $M \in SU(3)$, we first construct a general matrix D whose elements are complex numbers with both real and imaginary parts randomly sampled from the interval $[-1, 1]$. The matrix is then made Hermitian by applying the symmetrization:

$$H = \frac{1}{2}(D + D^\dagger). \tag{3.22}$$

Next, we construct a unitary matrix U by approximating the exponential of H using a truncated Taylor series expansion:

$$U = \sum_{k=0}^{n} \frac{(i\epsilon H)^k}{k!}. \tag{3.23}$$

In our implementation, we set $n = 30$ to ensure sufficient accuracy. The parameter ϵ controls the size of the update and must be tuned to achieve an acceptance rate of approximately 50% for trial updates. Following [2], we adopt the value $\epsilon = 0.24$.

The final step in constructing a pool of $SU(3)$ matrices is to enforce unit determinant. This is achieved by rescaling U as:

$$U \to \frac{U}{\det(U)^{1/3}}. \tag{3.24}$$

This procedure is implemented in the `SU(3)` function of our `WilsonLatticeUtils` class.

Listing 3.1: SU(3) Pool Generator

```python
def SU3(self, steps=30):
ones = (np.random.rand(3, 3)*2 - 1) + 1j *
(np.random.rand(3, 3)*2 - 1)
H = (1/2)*(ones + np.conj(ones.T))
U = np.zeros((3, 3), np.complex128)

for i in range(steps):
U += ((1j*self.eps)**i / np.math.factorial(i)) *
np.linalg.matrix_power(H, i)

return U / (np.linalg.det(U))**(1/3)
```

After creating a sufficiently large pool of $SU(3)$ matrices to adequately sample the group (in our case, 100 matrices plus their Hermitian conjugates), the next step is to construct the lattice on which the operations will be performed.[2]

[2]Logically, this should be the first step; however, we follow the order of functions in the class to help the reader better understand the code.

The lattice is initialized as a **7D empty array**, where four dimensions correspond to space-time coordinates, while the remaining three store a 3×3 identity matrix at each lattice site. These identity matrices represent the initial values of the link variables $U_\mu(x)$, which are updated according to the Metropolis algorithm.

The update of the lattice (implemented in the `lattice_update()` method) proceeds as follows: we pick a specific site x and a specific direction μ, then update the corresponding link variable $U_\mu(x)$ by multiplying it by a randomly selected matrix from the $SU(3)$ pool.

Listing 3.2: Lattice Update

```
n_hits = 10  # Number of Metropolis hits per link update

for t in range(self.N):
for x in range(self.N):
for y in range(self.N):
for z in range(self.N):
for mu in range(self.dim):  # Iterate over lattice dimensions
coord = [t, x, y, z]

# Compute fundamental plaquette term
gamma = self.plaquette(lattice, coord, mu)

# Compute improved rectangle term if needed
if self.use_improved:
gamma_imp = self.plaquette_improved(lattice, coord, mu)

for _ in range(n_hits):  # Perform Metropolis hits
xi = np.random.randint(2, self.N_matrix * 2)
U_old = lattice[coord[0], coord[1], coord[2], coord[3], mu]
U_new = np.dot(Xs[xi], U_old)

if self.use_improved:
dS = -self.beta_imp / 3 * (
(5 / (3 * self.u0**4)) *
np.real(np.trace(np.dot(U_new - U_old, gamma)))
- (1 / (12 * self.u0**6)) *
np.real(np.trace(np.dot(U_new - U_old, gamma_imp))))
else:
dS = -(self.beta / 3) *
np.real(np.trace(np.dot(U_new - U_old, gamma)))

# Accept or reject the update
if dS < 0 or np.exp(-dS) > np.random.uniform(0, 1):
lattice[coord[0], coord[1], coord[2], coord[3], mu] = U_new
```

This operation leads to a variation in the Wilson action given by:

$$\Delta S(x, \mu) = \Re\{\mathrm{Tr}\,(U_\mu(x)\Gamma_\mu(x))\}, \tag{3.25}$$

where $\Gamma_\mu(x)$ is obtained by summing the staples in all directions orthogonal to μ. A

staple is the product of the three link variables forming a plaquette, of which one side is the link which is being updated.

Typically, about 10 iterations ("hits") of the Metropolis algorithm are performed before moving to the next link variable. This ensures that the updated link variable reaches thermal equilibrium with its neighboring links. This operation is computationally inexpensive and represents another key difference compared to the one-dimensional case.

The explicit form of the action variation depends on the choice of action. We consider the two previously introduced cases:

1. Wilson action (Eq. (3.14)): In this case, only the plaquette operator $P_{\mu\nu}$ is needed, leading to the variation:

$$\Delta S(x,\mu) = \frac{\beta}{3u_0^4}\Re\{\mathrm{Tr}\,(U_\mu(x)\Gamma_\mu(x))\}. \tag{3.26}$$

2. Improved Wilson action (Eq. (3.9)): Here, we must also consider the rectangular plaquette operator $R_{\mu\nu}$, which is constructed using the function `plaquette_improved()`. The corresponding action variation becomes:

$$\Delta S_{\mathrm{imp}}(x,\mu) = -\frac{\beta_{\mathrm{imp}}}{3}\left\{\frac{5}{3u_0^4}\Re\left\{\mathrm{Tr}\,(U_\mu(x)\Gamma_\mu(x))\right\} - \frac{1}{12u_0^6}\Re\left\{\mathrm{Tr}\,\left(U_\mu(x)\Gamma^R_\mu(x)\right)\right\}\right\}. \tag{3.27}$$

By incorporating additional contributions from non-planar loops, one could achieve a more accurate lattice action. However, as can be observed directly from the code, larger loops require significantly more computational steps. A hypothetical `plaquette_big_loop()` method would involve an increasing number of calculations, making it computationally prohibitive.

The values of the plaquette ($a \times a$) and the rectangular Wilson loop ($2a \times a$) are evaluated in `WilsonLoop_main`, after performing the thermalization of the lattice through the function `lattice_update()` for `N_cor = 50` iterations. The results obtained by assuming the values suggest in [2] are shown in Table 3.1.

The results obtained are in perfect agreement with the expected value indicated by Professor LePage.

	Unimproved a×a	Unimproved a×2a	Improved a×a	Improved a×2a
1	0.503784	0.258365	0.540964	0.277024
2	0.493801	0.258380	0.536787	0.283862
3	0.493650	0.255885	0.545100	0.283298
4	0.492839	0.262779	0.536370	0.278848
5	0.494994	0.258511	0.541172	0.283591
6	0.490584	0.260499	0.539265	0.279716
7	0.490684	0.256736	0.540102	0.285113
8	0.494290	0.261076	0.537327	0.276410
9	0.491490	0.264385	0.539425	0.281266
10	0.496132	0.250426	0.542798	0.278091
11	0.494745	0.258308	0.539752	0.281990
12	0.501878	0.254891	0.535188	0.284984
13	0.498181	0.252966	0.538153	0.279188
14	0.497049	0.261698	0.538204	0.279462
15	0.499167	0.262179	0.542197	0.281771
16	0.492909	0.263585	0.541197	0.284675
17	0.487238	0.252073	0.540721	0.276737
18	0.492563	0.259364	0.536465	0.278755
19	0.499326	0.256660	0.538814	0.279148
20	0.494206	0.254229	0.539708	0.284301
Mean	0.494975	0.258150	0.539485	0.280911
σ	0.004015	0.003929	0.002439	0.002927

Table 3.1: Comparison of $a \times a$ and $a \times 2a$ Wilson Loop expectation value for Unimproved and Improved Wilson Action.

3.2 Static Quark Potential

As anticipated at the beginning of this last chapter, LQCD is particularly useful in studying confinement. The last part of our analysis consists indeed in the evaluation of the Static Quark Potential, i.e. the potential energy between a static quark and a static antiquark separated by a distance r. There are two ways one can test whether a theory confines. The first is to demonstrate that the free energy of an isolated charge is infinite; the second is to show that the potential energy between two charges grows with distance. The first test shows that it takes infinite energy to add an isolated test charge to the system, while the second shows that it requires infinite energy to separate two charges by infinite distance. The extra action associated with an external charge placed in a gauge field is

$$S_J = \int d^4x J^k_\mu(x) b^k_\mu. \tag{3.28}$$

For a point charge $J(x) = \delta^{(4)}(x)$ holds and SJ is given by the path ordered integral of the gauge field along the world line of the charge. Thus

$$\langle W \rangle = e^{S_J} \sim e^{-V(r)T} \tag{3.29}$$

where we indicated $\langle W \rangle$ as the expectation value of a $r \times T$ rectangular Wilson Loop . Thus

$$V(r) = \lim_{T\to\infty} -\frac{1}{T} \log W(r,T). \tag{3.30}$$

The simplest *Ansätz* for a confining potential is the Cornell potential

$$V(r) = \sigma r - \frac{b}{r} + V_0, \tag{3.31}$$

where σ is the string tension. At large distances σr dominates, while at short distances it is the Coulomb term b/r that is relevant. Such a potential, therefore, simultaneously exhibits confinement and asymptotic freedom. From Eq. (3.29) is also evident that one can calculate the static potential by computing $W(r,T)$'s for a variety of T's and then taking the large T limit

$$aV(r) \approx \frac{W(r,T)}{W(r,T+a)}. \tag{3.32}$$

3.2.1 Static Quark Potential: Code Implementation

Many of the methods already described in the analysis of the `WilsonLoopUtils` class are also fundamental for the evaluation of the Static Quark Potential. Therefore, to study this potential efficiently, we implemented a subclass
(`StaticPotentialUtils`) 4.5 of the aforementioned class, allowing all necessary methods to be inherited without needing to rewrite them. Among these inherited methods,

lattice creation, thermalization, and updating remain unchanged. Nonetheless, some key differences exist between the two implementations. Firstly, we introduce a modification that significantly improves the results: replacing the spatial link matrices in the direction of separation (r direction) with "smeared" link variables $\tilde{U}\mu(x)$, defined by

$$\tilde{U}_\mu(x) \equiv (1 + \epsilon_{\text{sm}} a^2 \Delta^{(2)})^n U_\mu(x), \tag{3.33}$$

where

$$\Delta^{(2)} U_\mu(x) = \sum_\rho \Delta^{(2)}_\rho U_\mu(x) \tag{3.34}$$

is the gauge covariant-derivative implemented in the method of the same name and defined as

$$\begin{aligned}\Delta^{(2)}_\rho U_\mu(x) \equiv & \frac{1}{u_0^2 a^2} (U_\rho(x) U_\mu(x + a\rho) U^\dagger_\rho(x + a\mu) - 2u_0^2 U_\mu(x) \\ & + U^\dagger_\rho(x - a\rho) U_\mu(x - a\rho) U_\rho(x - a\rho + a\mu)).\end{aligned} \tag{3.35}$$

Here, ϵ_{sm} is a small parameter associated specifically with the smearing procedure, distinguished from the ϵ parameter used in the Monte Carlo algorithm.
The smearing procedure is crucial as it acts as a momentum cutoff, suppressing high-momentum gluons:

$$(1 + \epsilon_{\text{sm}} a^2 \Delta^{(2)})^n \quad \rightarrow \quad (1 - \epsilon_{\text{sm}} a^2 p^2)^n \approx e^{-\epsilon_{\text{sm}} a^2 p^2} \tag{3.36}$$

while leaving the low-momentum components of the smeared link variables unaffected. Additionally, it accelerates the convergence of the ratio $W(r, T)/W(r, T + a)$. The smearing algorithm is implemented in the `smearings()` method according to Eq. (3.33), and the `smear_lattice()` method applies it repeatedly (`number_of_smears = 4` in our case) to the entire lattice.

Listing 3.3: Smearing

```
def smear_lattice(self, lattice):
for t in range(self.N):
for x in range(self.N):
for y in range(self.N):
for z in range(self.N):
point = np.array([t, x, y, z])
for direction in range(1, self.dim):
smeared_link = lattice[t, x, y, z, direction]
+ self.smeared_eps * (self.a ** 2)
* self.gauge_covariant_derivative(lattice, point, direction)

#SU(3) projection process
U, S, Vh = np.linalg.svd(smeared_link)  # SVD
lattice[t, x, y, z, direction] = np.dot(U, Vh)
```

```
detU = np.linalg.det(lattice[t, x, y, z, direction])
lattice[t, x, y, z, direction] /= detU**(1/3)

return lattice

def smearings(self, lattice, number_of_smears):
repeatedly_smeared_lattice = lattice.copy()
for i in range(number_of_smears):
repeatedly_smeared_lattice =
self.smear_lattice(repeatedly_smeared_lattice)

return repeatedly_smeared_lattice
```

Another peculiarity of the `smear_lattice()` method is the presence of a projection step for the link variables. Although the smearing procedure is highly beneficial for the reasons previously stated, it does not map $SU(3)$ matrices into matrices belonging to the same group; thus, a projection of these matrices back onto $SU(3)$ is required. Several methods exist for performing this projection. The method we chose proceeds as follows:

1. The function `np.linalg.svd()` performs a Singular Value Decomposition (SVD) of the `smeared_link` matrix:

$$M = U\Sigma V_h^\dagger, \tag{3.37}$$

   where U and $V_h^\dagger$ are 3×3 unitary matrices, and Σ is a diagonal matrix containing real, positive entries (known as singular values).
2. The operation `np.dot(U, Vh)` yields a unitary matrix, as it is the product of two unitary matrices.
3. The determinant is then set to unity by applying Eq. (3.24).

Such a projection ensures that the resulting matrix is the "closest" possible to the original one while still belonging to the $SU(3)$ group.
Another key difference arises in the implementation of the `planar_loop()` method, which is the only method present in both classes. This difference occurs because, to correctly evaluate the Static Quark Potential, the loop must always be oriented along one spatial (in order to extract quark separation) and one temporal dimension(in order to extract the energy). Consequently, loops oriented purely in spatial directions (e.g., the xy-plane) or purely in temporal directions must not be considered, as they would not yield the correct result.
The data generated by the `run_simulation()` method, which performs both the lattice thermalization and the evaluation of Wilson loops along with their associated uncertainties, are subsequently processed by the `analyze_results()` method. This method computes the static quark potential using Eq. (3.32), clearly distinguishing between the smeared and non-smeared cases. Finally, the obtained data are fitted according to

the Cornell potential defined in Eq. (3.31). The obtained plots for the Static Quark Potential for are shown below.

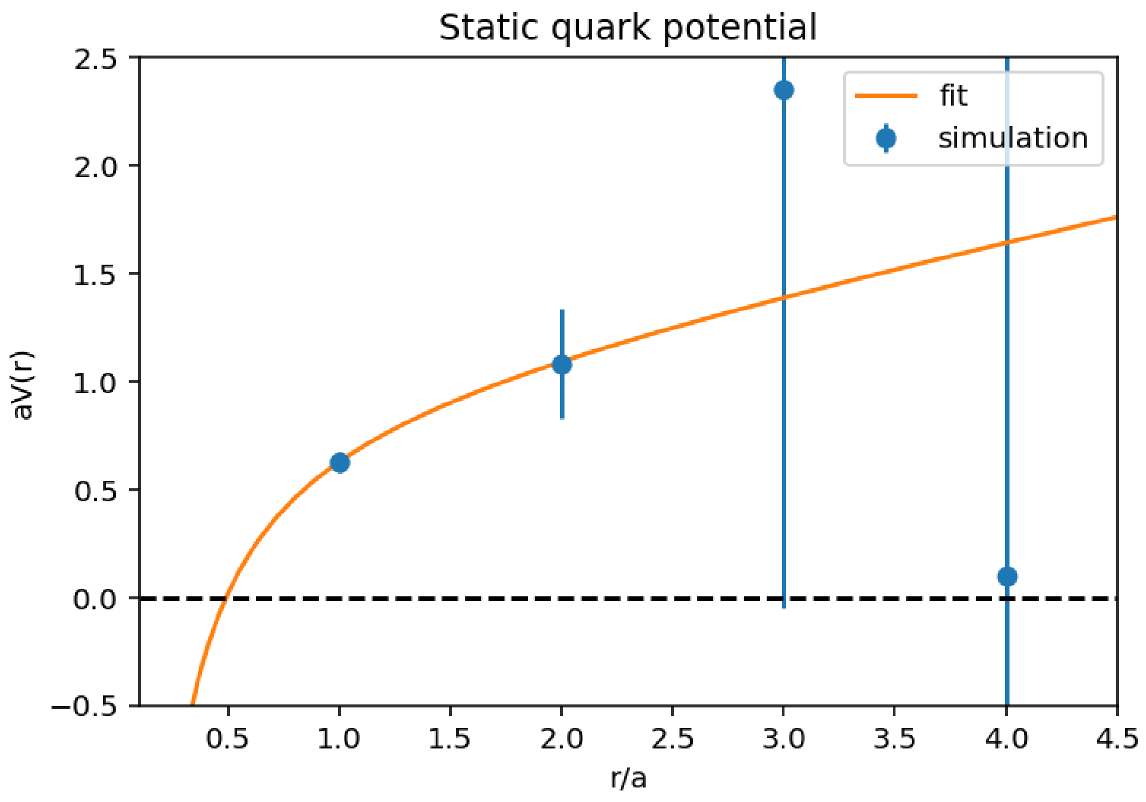


Figure 3.1: Static Quark Potential with Unimproved Action and Unmeared Links

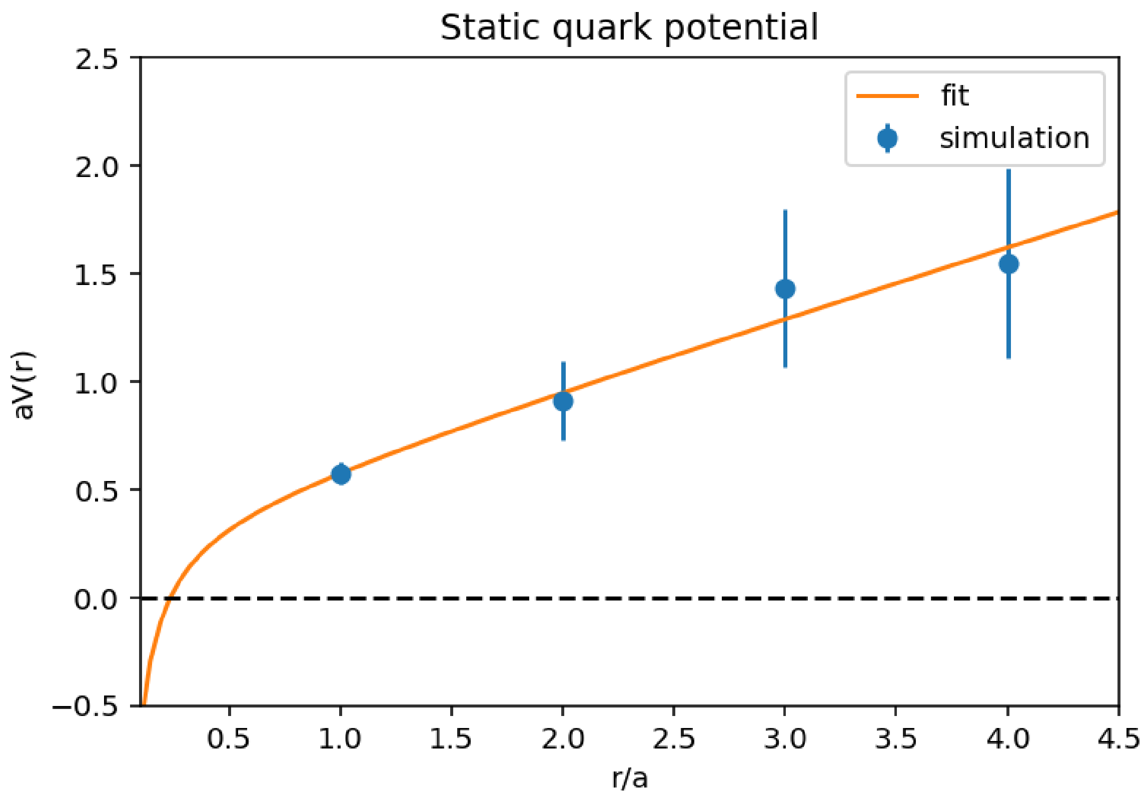


Figure 3.2: Static Quark Potential with Unimproved Action and Smeared Links

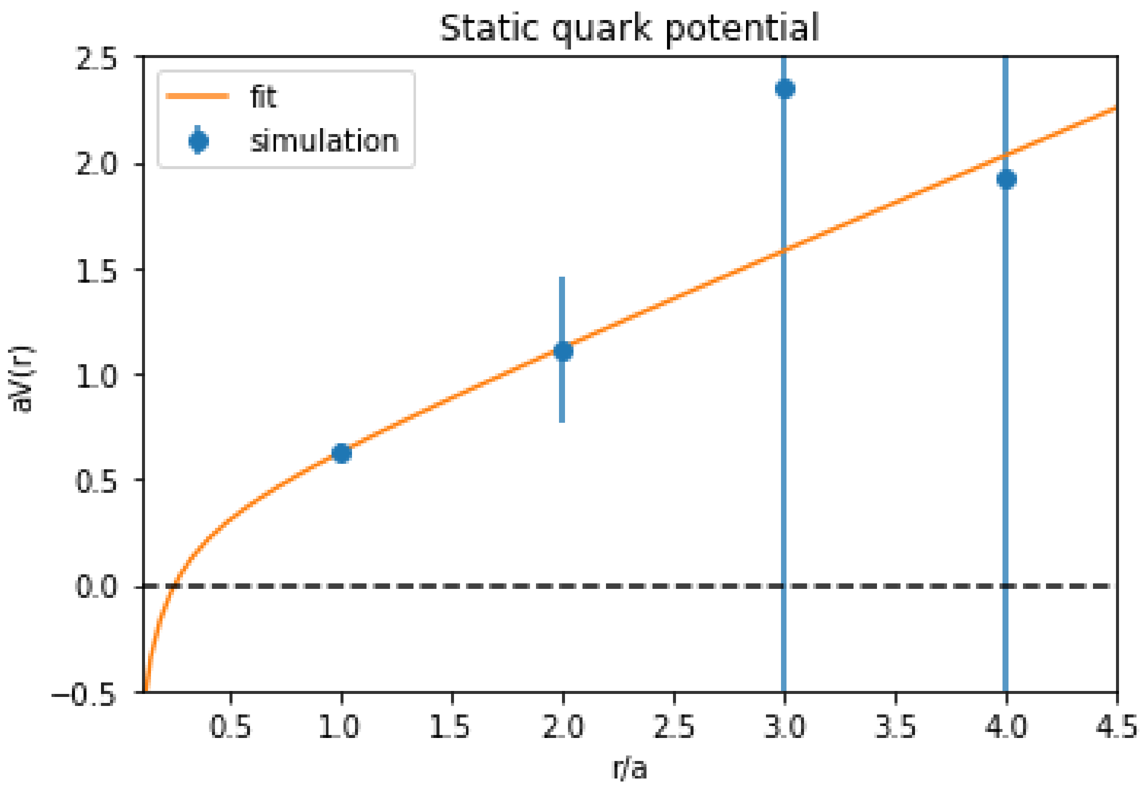


Figure 3.3: Static Quark Potential with Improved Action and Unsmeared Links

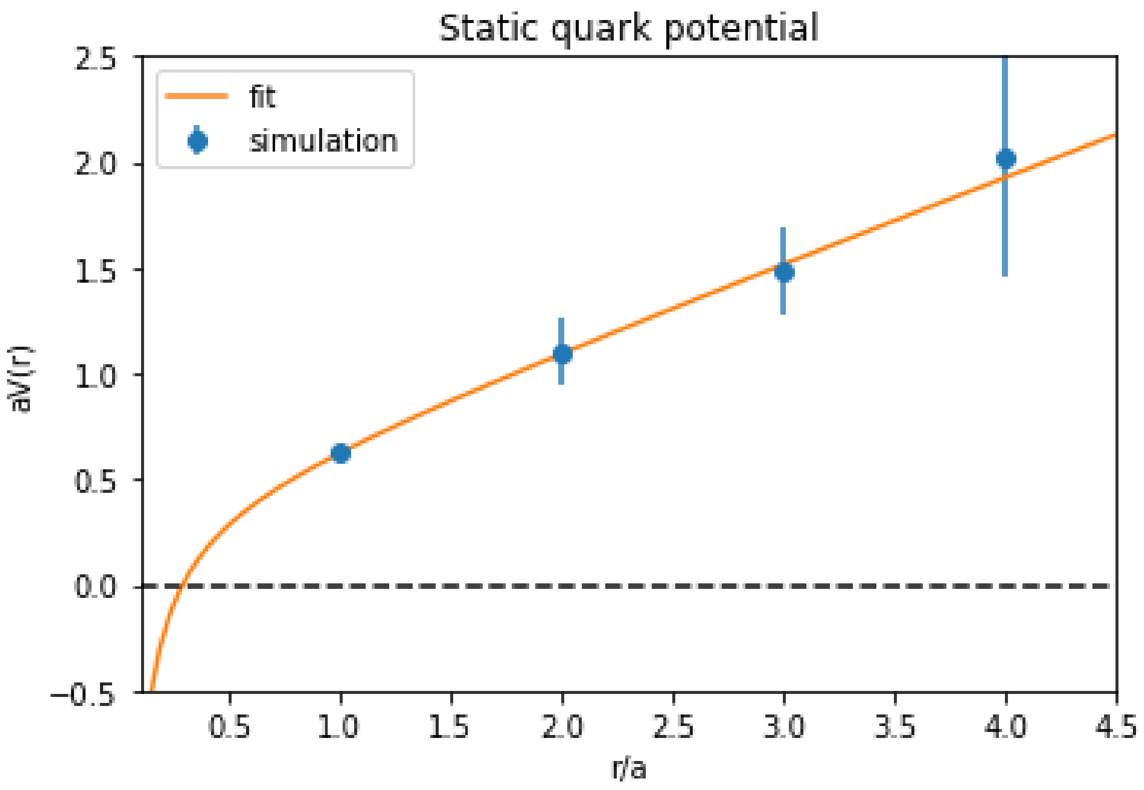


Figure 3.4: Static Quark Potential with Improved Action and Smeared Links

Before commenting on the results, we also report the values obtained for the free parameters (σ, b, and V_0):

Improved action	**Smearing**	**Parameters Fit values**
No	No	$\sigma = 0.2124(7) \pm 1.2342(6)$ $b = 0.5007(6) \pm 2.5241(1)$ $V_0 = 0.9161(3) \pm 3.7532(6)$
No	Yes	$\sigma = 0.3234(7) \pm 0.1595(7)$ $b = 0.1000(0) \pm 0.4465(3)$ $V_0 = 0.3499(2) \pm 0.5980(6)$
Yes	No	$\sigma = 0.4435(5) \pm 0.8645(2)$ $b = 0.1000(0) \pm 1.7413(6)$ $V_0 = 0.2839(4) \pm 2.6048(4)$
Yes	Yes	$\sigma = 0.3999(4) \pm 0.0800(0)$ $b = 0.1400(6) \pm 0.2165(1)$ $V_0 = 0.3627(9) \pm 0.2943(2)$

Table 3.2: Fit results for the values of the parameters for the Static Quark Potential. All studied cases are included

As one can see from both Table 3.2 and the plots, the smearing procedure significantly reduces the errors associated with the parameters. In the unsmeared case, these errors are considerably larger than the values of the parameters themselves, making the results unreliable. This is particularly important because the magnitude of the errors increases significantly as the size of the considered loops grows. Thus, to ensure the convergence of the results, the smearing procedure is essential.

Even in the smeared case, however, we were not able to achieve fully satisfactory results, as in some cases the errors remain too large. The origin of this issue is not entirely clear. One possible explanation is the highly fluctuating behavior observed in the evaluation of the largest loops, which amplifies the uncertainty beyond control. Nonetheless, for smaller values of r/a, the error bars are very small, suggesting a good degree of reliability in the extracted potential. In general, the potential exhibits the expected behavior for all the considered cases: as anticipated, we observe $V(r) \propto r$, which confirms the presence of confinement.

Chapter 4

Appendix

4.1 NumericalPathIntegral.py

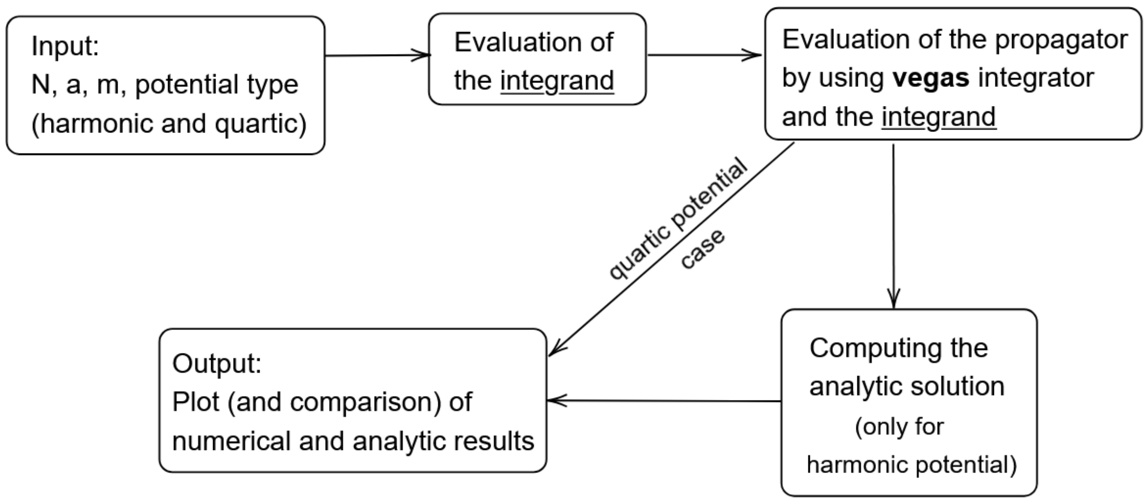


Figure 4.1: Logical flow of the `NumericalPathIntegral.py` Class

4.2 PathIntegralMonteCarlo.py

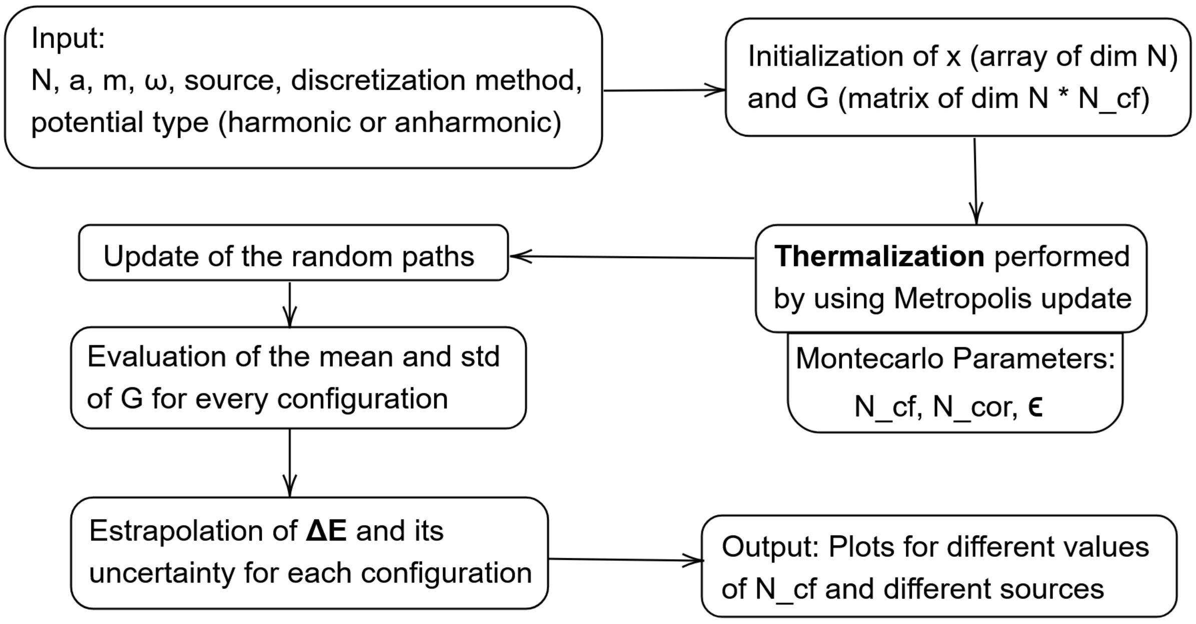

Input:
N, a, m, ω, source, discretization method, potential type (harmonic or anharmonic)
Initialization of x (array of dim N) and G (matrix of dim N * N_cf)
Thermalization performed by using Metropolis update
Montecarlo Parameters: N_cf, N_cor, ϵ
Update of the random paths
Evaluation of the mean and std of G for every configuration
Estrapolation of ΔE and its uncertainty for each configuration
Output: Plots for different values of N_cf and different sources

Figure 4.2: Logical flow of the `PathIntegralMonteCarlo.py` Class

4.3 EOMSolver.py

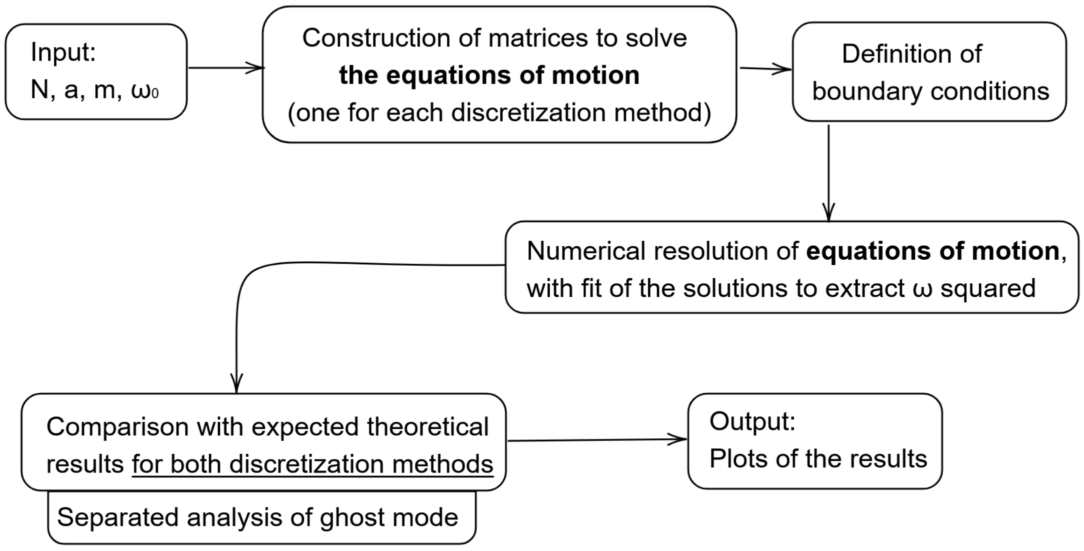

Input:
N, a, m, ω₀
Construction of matrices to solve **the equations of motion** (one for each discretization method)
Definition of boundary conditions
Numerical resolution of **equations of motion**, with fit of the solutions to extract ω squared
Comparison with expected theoretical results for both discretization methods
Separated analysis of ghost mode
Output:
Plots of the results

Figure 4.3: Logical flow of the `EOMSolver.py` Class

4.4 WilsonLatticeUtils.py

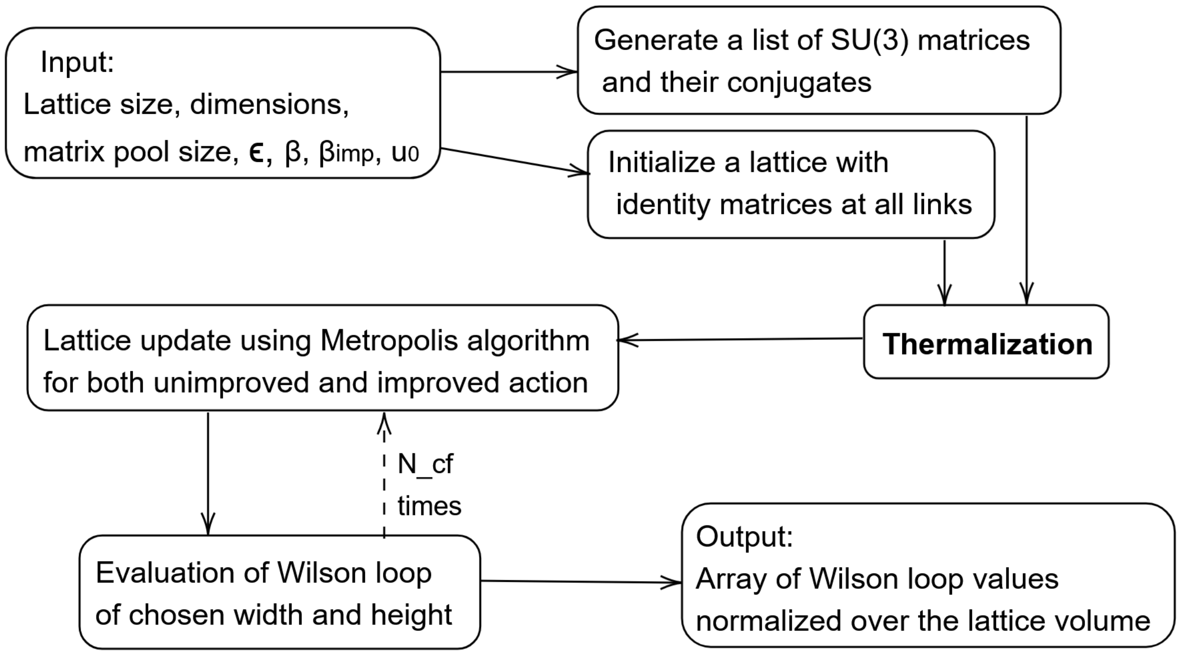
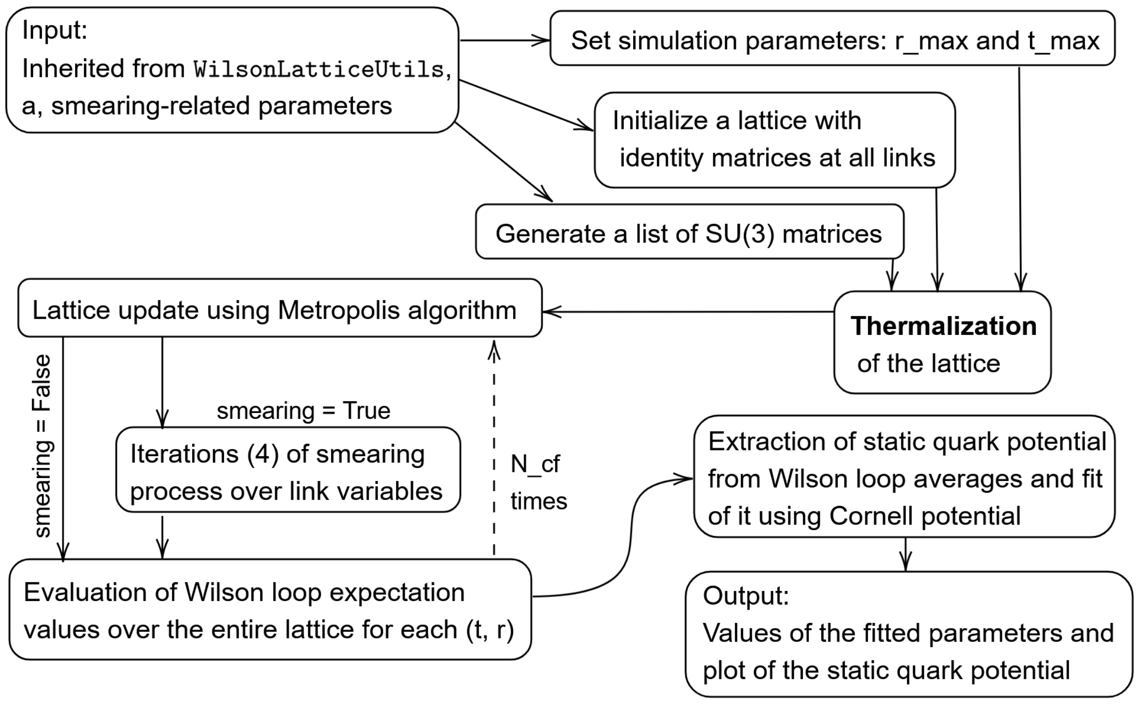

Input:
Lattice size, dimensions,
matrix pool size, ϵ, β, βimp, u0
Generate a list of SU(3) matrices and their conjugates
Initialize a lattice with identity matrices at all links
Thermalization
Lattice update using Metropolis algorithm for both unimproved and improved action
N_cf times
Evaluation of Wilson loop of chosen width and height
Output:
Array of Wilson loop values normalized over the lattice volume

Figure 4.4: Logical flow of the `WilsonLatticeUtils.py` Class

4.5 StaticPotentialUtils.py



Input:
Inherited from WilsonLatticeUtils,
a, smearing-related parameters
Set simulation parameters: r_max and t_max
Initialize a lattice with identity matrices at all links
Generate a list of SU(3) matrices
Thermalization of the lattice
Lattice update using Metropolis algorithm
smearing = False
smearing = True
Iterations (4) of smearing process over link variables
N_cf times
Evaluation of Wilson loop expectation values over the entire lattice for each (t, r)
Extraction of static quark potential from Wilson loop averages and fit of it using Cornell potential
Output:
Values of the fitted parameters and plot of the static quark potential

Figure 4.5: Logical flow of the `StaticPotentialUtils.py` Class

Bibliography

[1] F. Marini, R. Bianconcini, *An Introduction to Lattice QCD*, 2025.
GitHub Repository:
`https://github.com/Mar-physics/An-introduction-to-Lattice-QCD-`.

[2] G. P. LePage, *Lattice QCD For Novices*, arXiv, 2005.
https://arxiv.org/abs/hep-lat/0506036

[3] R. Feynmann, *Space-Time Approach to Non-Relativistic Quantum Mechanics*, Rev. Modern Physics. 20, 1948.

[4] R. Gupta, *Introduction to Lattice QCD*, arXiv, 1997.
https://doi.org/10.48550/arXiv.hep-lat/9807028