Computational Physics — Semester 2 Project

Simulating Quark Deconfinement using Lattice QCD

Tim Daly

BSc Theoretical Physics and Pure Mathematics

Date of Submission:
June 9, 2025

Abstract

This project investigates the confinement–deconfinement phase transition in Quantum Chromodynamics (QCD), focusing on the conditions under which quarks become deconfined at high temperatures. A simplified model of the strong interaction is implemented using Monte Carlo simulations within the framework of lattice QCD. The study begins with pure gluodynamics and examines how to incorporate quarks to explore the emergence of free quarks. Simulations are conducted using an SU(2) gauge group to represent a two color theory, serving as a building block to the full SU(3) theory. Observables such as the average plaquette and the Polyakov loop are used to identify the critical temperature and examine the conditions under which this phase transition occurs. The results provide insight into the workings of deconfinement and lay the groundwork for future studies involving more realistic gauge groups and dynamical fermions.

Contents

1 Strong Force

Nearly all of the matter we see is made up of quarks. But a free quark has yet to be observed. This is due to the strong force, one of the four fundamental forces, that binds quarks together to form particles such as the proton and neutron. It also holds the nucleus together and underpins interactions between any particles containing quarks.

Quarks and gluons are color-charged particles analogous to how particles can be electrically charged and they interact by exchanging photons. Color-charged particles interact by exchanging gluons, this is what the strong force is. If we have two or more quarks close to each other they rapidly exchange gluons which creates a strong 'color force field' which binds them together.

There are three color charges and three corresponding anti-color charges which anti-quarks possess. Namely red,green, blue and rather unimaginatively anti-red,anti-green and anti-blue. As nearby quarks are constantly exchanging gluons their color charge also changes. Color charge is a conserved quantity, so when a quark either absorbs or emits a gluon, it's color must change to conserve the overall color charge.

1.1 Confinement

The rapid exchange of gluons between quarks is referred to as the color-force field. If we try to pry one the quarks away from its partners then this force field 'stretches'[13]. As we keep pulling it away, more and more energy is added to this force field until at some point it becomes more energetically efficient for this field to snap (shown in 1) and convert this energy into the creation of a quark-antiquark pair which then bind to the original quarks. This is the quark confinement of which we speak and is why it is nearly impossible to separate quarks.

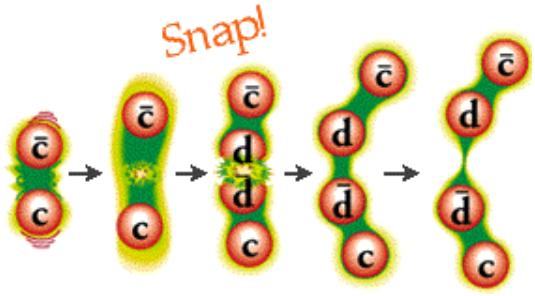

Figure 1: Depiction of quark confinement[13]

At extremely high temperatures, about $10^9 K$, the quarks gain enough energy to escape this confinement and they can form Quark-Gluon plasma. This is a state of matter where quarks and gluons are free to move about. The phase transition from confinement into a quark gluon plasma (deconfined) as seen in the graphic below is what we are interested in.

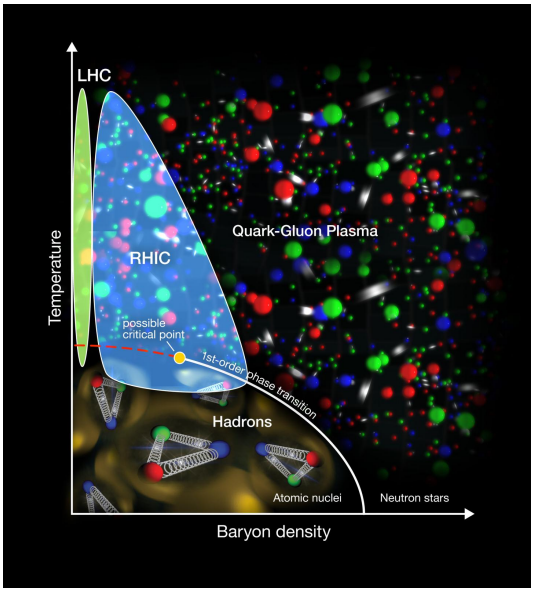

Figure 2: QCD phase diagram from Brookhaven National Laboratory

The above QCD phase diagram, illustrates the state of matter as a function of temperature and baryon density. At low temperatures and densities, quarks are confined within hadrons, while at extremely high temperatures like those created at the LHC and RHIC, matter transitions into a quark-gluon plasma, where quarks and gluons are deconfined. The first-order phase transition line marks the boundary between hadronic matter and the deconfined phase.

2 QCD

Quantum Chromodynamics or QCD is the quantum field theory that underpins the strong interaction. Strongly interacting particles are called hadrons, made up of quarks and gluons, which are the underlying constituents in QCD. The quarks interact strongly through their "color" charge by exchanging gluons. There are 8 types of gluons and they carry colour, this means they can interact with themselves (gluodynamics) as well as with quarks. It is a non-abelian gauge theory which is invariant under SU(3). Many of the analytic results are difficult to find and are quite complicated. A way around this is to use lattice QCD. We discretise time and space onto a lattice allowing us to simulate interactions between quarks and gluons in a non-perturbative manner.

3 Gauge Theory

Choice of gauge and formulating a gauge theory on the lattice is an important aspect of lattice quantum field theories. Here i give a brief overview of the relevant background. I found *David Tong's* notes on Gauge theory[9] to be the most useful and most of what I layout in this section are his notes which I supplement with further clarifications if needed.

A **Gauge** refers to a particular choice or configuration of the mathematical functions or fields that represent redundancies in the description of the Lagrangian. Gauge transformations are what we call transformations between possible gauges, these transformations form a Lie group. Associated with this Lie group is another group which form the gauge field components.

The gauge field is sometimes referred to as a connection, which gives us some insight into what it actually is. In our case dealing with quarks and gluons, the gauge field acts as a guide, instructing the color degrees of freedom how to evolve, essentially determining how they interact. We call it a connection since it 'connects' these color degrees of freedom at one point in space to another, so from one gluon or quark to another.

The gauge field is formed using Lie valued components A_μ, as the fundamental objects. For our formalism on a lattice we will use group valued variables, namely SU(2) variables U_μ to replace these A_μ. These are what will tell other fields how to evolve. These links also called Wilson lines, lie between sites while the matter fields live on the sites of the lattice. We explain in 5 further how these links U_μ actually relate to direction and specific sites on the lattice. In the scope of the gauge theory all we need to know is that we have taken these Lie algebra valued objects and are now representing them using group valued objects. This switch is a central idea in lattice gauge theory

We often want to go between the lattice gauge and the continuum action, we do so by reintroducing the A_μ gauge fields,

$$U_\mu(x) = e^{iaA_\mu(x)}$$

On our lattice we need to know how these link variables transform, it does so as

$$U_\mu(x) \to \Omega(x)U_\mu(x)\Omega(x+\mu)$$

The whole idea of gauge theory hinges on the action of the system. The classical equations of motion are derived by minimising the action with respect to each gauge field A_μ. So we need a lattice version of the action.

First we need the concept of a **gauge symmetry**. It is a type of symmetry where the physical laws remain invariant under certain local transformations. These transformations are associated with a Lie group which represent the symmetry groups of the interactions.

Gauge symmetry is a bit of a misnomer, it isn't a symmetry in the classical sense. In this case it represents a redundancy in our description of the system. This redundancy is actually a useful way of describing physical systems, as it give us a way to describe the dynamics of the theory in a way that makes manifest various properties of the theory such

as Lorentz invariance or in quantum theory the unitarity[9].

This redundancy that is inherently embedded in the definition of gauge symmetry means that only gauge invariant quantities should be considered physical. That is things that don't rely on the description. We can think of these as akin to 'coordinate independent' quantities.

Now that we know physical observables need to be gauge invariant we search for a description of our action on the lattice in terms of gauge invariant quantities. The principal of which will be the Wilson loop.

3.1 Wilson Loop

A **Wilson loop** is defined as,

$$W(C) \equiv \mathrm{Tr}\left(\mathcal{P}\prod_{x\in C} U_\mu(x)\right)$$

where C is any closed path and $\mathcal{P}$ is the **path ordered product** which is a way of multiplying the gauge links (matrices) along a path in a specific order that reflects the direction of motion. This stipulation is important for non-abelian gauge theories where matrix multiplication is not commutative. The trace here ensures the gauge invariance. These so called Wilson loops represent closed paths C around our lattice. The simplest of which is a square that traverses one lattice site for each side of the square returning to the start. This is the **plaquette** avery useful quantity and one we use alot, defined as

$$P_{\mu\nu}(x) = \frac{1}{2}Tr\left(U_\mu(x)U_\nu(x+\hat{\mu})U_\mu^\dagger(x+\hat{\nu})U_\nu^\dagger(x)\right) \tag{1.1}$$

summing over all possible plaquettes gives us the action we were after,

$$S = \frac{\beta}{N}\sum_{i=1}^{6} \mathrm{Re\ Tr}\left(I - U_\mu(x)S_\mu(x)\right)$$

where $S_\mu(x)$ are the staples, I explain what these are in 8.2 in more detail but they simply represent taking the sum over all possible plaquettes.

4 SU(2) vs SU(3)

In actuality as explained above, every quark comes in 3 colors and the gluons can be described by 3×3 unitary matrices. The 3×3 and 2×2 unitary matrices form the SU(3) and SU(2) groups respectively.

A quick reminder the SU(N) group is the Lie group consisting of,

- $n\times n$ unitary matrices, i.e $UU^\dagger = I$
- With determinant 1

In this project I study quarks that only come in 2 colors, and the gluons can be represented by 2×2 unitary matrices. The 2-color and 3-color theory share all the same essential features, including confinement at low temperatures and deconfinement at high temperatures. The

main advantage is that the SU(2) matrices are much easier to calculate with. All the algorithms presented here can be altered 'easily' enough to handle the SU(3) case.

4.0.1 Pauli representation

The first thing about SU(2) to note is that all SU(2) matrices can be written in terms of 4 real numbers $u_0 ... u_3$

$$U = u_0 I + i\vec{u} \cdot \vec{\sigma} \qquad u_0^2 + u_1^2 + u_2^2 + u_3^2 = 1$$

where σ are Pauli matrices. So the matrix U looks like

$$U = \begin{pmatrix} u_0 + iu_3 & iu_1 + u_2 \\ iu_1 - u_2 & u_0 - iu_3 \end{pmatrix}$$

The constraint that the square of the four numbers must add to 1 is so that the matrices are unitary. If we take the determinant of the above U

$$det(U) = u_0^2 + u_1^2 + u_2^2 + u_3^2$$

=1 this needs to be 1 to be in SU(2). So we use these four numbers (u_1, u_2, u_3, u_4) to fully represents any SU(2) matrices. For SU(3) we need 18 real numbers to represent elements.

The trace of these matrices is an important value but since the Pauli matrices are traceless we see the trace is simply

$$Tr(U) = 2u_0$$

5 Navigating the Lattice

As mentioned before we will be discretising time and space onto a lattice, specifically a 4-dimensional grid with 3 spatial dimensions and one temporal. The distance between points on the lattice is called the lattice spacing β, and loosely represents the temperature,more about this in 11.2. In this set up the quarks are represented by the lattice sites and the gluons live on the links between the sites*.

We work with what is called a hypercubic lattice, which generalises the familiar 3D cubic grid to higher dimensions, hypercubic lattices have the following properties,

- They are symmetric in all directions, so each point has the same number of neighbouring points.
- The lattice points are evenly spaced, and each point is connected to its nearest neighbours by edges or what we call link variables.
- For an n-dimensional lattice, each point has $2n$ nearest neighbours.

*This configuration leads to quite an unwieldy data structure as we have the 4 spacetime dimensions, each of these lattice sites thus has 4 link variables and each of these four link variables is represented by a vector of 4 real numbers. This can get a bit messy but there is not much to be done about it as it is the most efficient representation of such a system

5.1 Boundary Conditions

We use Λ to denote the lattice,

$$\Lambda = \{n = (n_1, n_2, n_3, n_4) | n_\mu = 0, 1, ...N_\mu - 1\}$$

where in our case we use $N_1 = N_2 = N_3 = N_\sigma, N_4 = N_T$, that is the the three spatial dimensions are equal and the temporal dimension is usually smaller. We will refer to the size of the lattice as $N_\sigma^3 \times N_T$, so $8^3 \times 4$ would represent a lattice with 8 sites in each spatial direction and 4 sites in the time direction.

Since we are working with a numerical simulation we will be using a finite lattice, so we have to employ boundary conditions. For gauge fields it is standard to use periodic boundary conditions;

$$U_\mu(N_1, n_2, n_3, n_4) = U_\mu(0, n_2, n_3, n_4)$$

and the same for the other dimensions.

Periodic boundary conditions correspond to a torus in four dimensions- each direction behaves like a circle looping back around to the 'start' of the lattice. Choosing these boundary conditions has the advantage of preserving discrete translational symmetry of the lattice[7].

On a computational note, we often need the indexes of the nearest neighbours to a point, for example in the calculation of the the change in action ΔS. During calculations this actually takes up a non-trivial amount of computation time and so various techniques have been created to accelerate this. One way is using pre-calculated index arrays which help implement nearest neighbour calculations at the same time as boundary conditions.

We denote the link variable between a point on the lattice x and the neighbouring point $(x + \hat{\mu})$ in the μ- direction by $U_\mu(x)$. The backward link from $x + \mu$ to x is $U_\mu^\dagger(x)$.

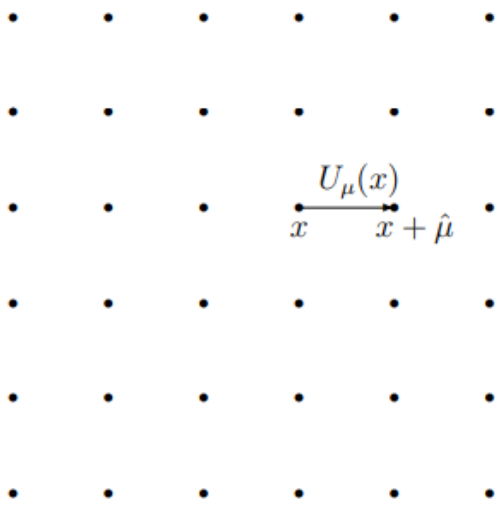

Figure 3: Lattice sites and link variables

5.2 The Plaquette operator

The shortest loop on our lattice is a step in some direction μ and then a step back, since all of our link variables are unitary we see,

$$U_\mu(x)U_\mu^\dagger(x) = I$$

This is just a check that our lattice is formulated correctly but is not very interesting. The shortest closed loop on a lattice that isn't the trivial case, turns out to be a very important quantity. We call it the *plaquette operator*. It is simply a square, formed by 4 lattice sites and their link variables in some plane $\mu \neq \nu$. It is defined as

$$P_{\mu\nu}(x) = \frac{1}{2}Tr\left(U_\mu(x)U_\nu(x+\hat{\mu})U_\mu^\dagger(x+\hat{\nu})U_\nu^\dagger(x)\right) \tag{1.2}$$

If we define our plaquette to run in a clockwise direction, we can form a counter-clockwise counterpart in the same way,

$$P_{\mu\nu}^{ccw} = U_\nu(x)U_\mu(x+\hat{\nu})U_\nu^\dagger(x+\hat{\mu})U_\mu^\dagger(x)$$

But it can easily shown through a quick calculation that in fact

$$P_{\mu\nu} = P_{\mu\nu}^{ccw}$$

so the orientation is not important. To prevent us from double counting we choose the clockwise orientation and then impose the restriction $\mu < \nu$ when calculating these plaquettes.

What we really want is the average plaquette over the whole lattice.

$$P[U] = \frac{1}{6N_{\text{lat}}}\sum_{\vec{x},\mu<\nu} P_{\mu\nu}(x) \tag{1.3}$$

The average plaquette is a very useful quantity for us. First and foremost it allows us to examine when our algorithm has thermalised, we will examine this in detail in section 6. It also allows us to perform a sanity check to make sure our algorithm is working as expected, as we can compare it to established results in the field. If we want some physical definition of the average plaquette, it represents the field strength of the gauge field on the lattice. So lets perform some simulations of this average plaquette on the lattice and see how it behaves.

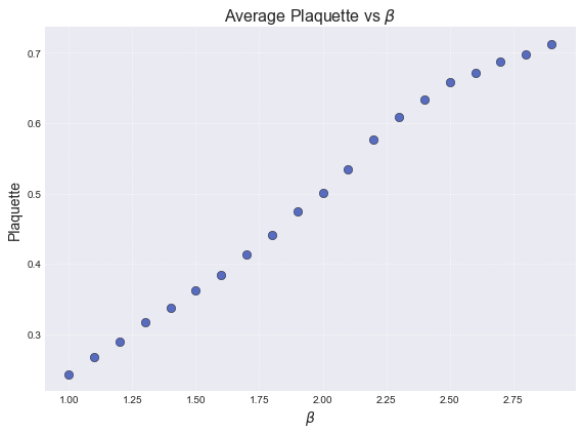

Figure 4: Mean average plaquette vs β from my simulation for a $4^3 \times 4$ lattice for β ranging from 1.0 to 3.0 in steps of 0.1

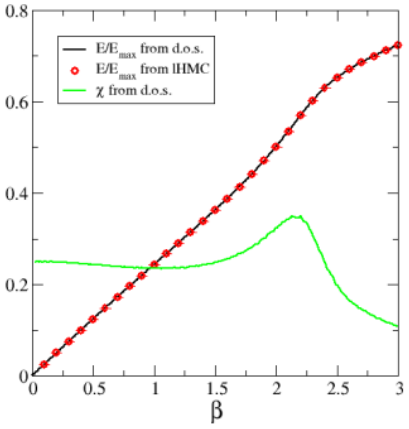

Figure 5: Reference plot from [7], the red plot is what we are trying to mimic.

In the above graph I compare the mean average plaquette value of my simulation for various values of β and compare it to known results from [8].We use a $4^3 \times 4$ lattice so we can avail of our results from 6 so we know it will have thermalised after about 20 sweeps so we can discard the first 20. We then calculate the average plaquette using (1.2). We store these values for the average plaquettes for each of the 200 sweeps we perform. At the end of these 200 sweeps we take the mean of these and that gives us our value for the average plaquette for that value of β. We continue this for the next β continuing in steps of 0.1 up to $\beta = 3$. We can see the graph is essentially linear up to about $\beta = 2.00$, then we get a slight kink in the graph exactly as shown in the known results. Our results very closely mimic the reported ones so we know our algorithm is working as expected.

We can see that somewhere between $\beta = 2.1$ and $\beta = 2.2$ is the maximum slope, this can be an indicator of a phase transition, possibly from the confined state to the deconfined state. The average plaquette gives us only a possible value for a phase transition, we require further analysis using the Polyakov loop which we do later in 10. It does however give us a good indication of where to look for one.

5.2.1 A note on orientation

It is important that we define our orientation for our lattice as it needs to be consistent throughout to avoid double counting. We have defined the plaquette to be clockwise, so $\mu < \nu$. We also need to define the orientation for the link variables. As noted before for a n-dimensional lattice, each site has $2n$ neighbours. So for our 4 dimensional lattice we has 8 neighbours. But we only need consider 4 of them, since if we decide we only take neighbours to the 'right' or equivalently direction vectors with positive components for (x, y, z, t) then the other four links will be picked up by the lattice site to the left of this one. This ensures we cover the entire lattice counting each link variable just once.

6 Thermalisation

After we have updated all of the link variables at every site once, we have completed one 'sweep'. As mentioned before we want to track the average plaquette value of the whole lattice after each sweep. Monitoring this we can determine when the lattice link distribution has reached equilibrium. For any fixed value of beta the average plaquette will start near zero and then converge to a roughly constant value. The number of iterations it takes to converge around this value is called the *thermalisation* or burn-in.

Before our algorithm has thermalised the lattice link distribution is not distributed in accordance with the desired probability. So we want to throw away any sweeps before the plaquette value has settled down.

In Figure 6 we can see how the average plaquette values asymptotically approach a constant value for various lattice sizes. We see this value is independent of lattice size, however the larger lattices experience less noise from the Monte Carlo method. We can see that for the $8^3 \times 4$ lattice it only takes about $n_{therm} = 20$ iterations to thermalise. So we discard n_{therm} sweeps and then we can take observables and measurements.

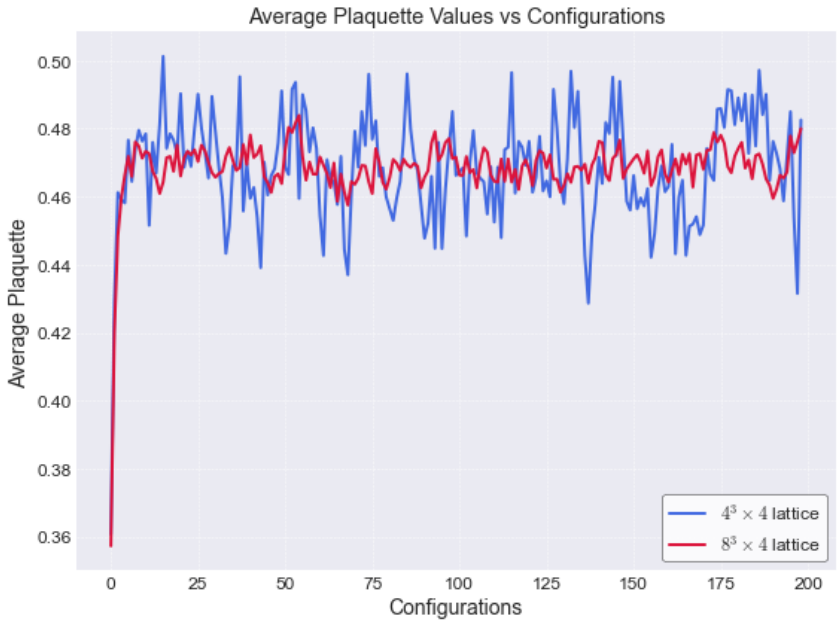

Figure 6: Average plaquette values on $4^3 \times 4$ and $8^3 \times 4$ lattices over 200 sweeps with $\beta = 1.9$.

6.1 Analysis of noise versus lattice size

As we can see in Figure 6 the larger the lattice size the less the average plaquette varies about this constant value. We call this the 'noise' of the Monte Carlo simulation and it is present whenever we use any sort of probabilistic Markov Chain model. We wish to try minimise this noise to enhance the accuracy of our results. Let us analyse exactly how this noise diminishes as we increase the lattice size.

First it is important we thermalise our lattice so we can get representative averages. From Figure 6 we know we only need about 20 sweeps to thermalise. We will use lattices with spatial dimensions ranging from $4^3 \times 4$ to $12^3 \times 4$. We perform 100 sweeps after thermalisation calculating the average plaquette after each sweep. For each lattice we want the mean as this is the constant value it should be approaching. What we are calling 'noise' is just how much the Monte Carlo method varies from this constant. Even though we are picking our random values so they approximate certain distributions this method is not perfect and leads to some small error interval. So in this case it is how much our average plaquette values vary from this constant. We can quantify this by calculating the standard deviation from the mean for each lattice size and seeing how this decreases as the lattice size increases. We can try to extract from this a relationship between the two.

We will use a $16^3 \times 4$ lattice and 100 sweeps to obtain a numerical value for this constant as this is largest lattice size feasible, and will give us a sufficiently accurate result, which is

$$P_{\text{avg}} \approx 0.4701$$

In the below plot we have graphed the standard deviation for various lattice sizes. We would like to be able obtain a function from this that gives us exactly how the noise of the Monte Carlo method decreases as we increase the lattice size. We expect it to be some sort of inverse square root function, usually with respect to the lattice volume as is typical of lattice scaling in lattice theories. Say we try

$$\sigma = \frac{1}{\sqrt{V}}$$

where σ, the standard deviation is our 'noise' and $V = N_x \times N_y \times N_z \times N_T$. We see this has a similar shape to the graph but we need a constant factor to scale it correctly. So our job now lies in finding this constant. This is pretty straightforward, simply use

$$\sigma = \frac{\alpha}{\sqrt{V}} \implies \alpha = \sigma\sqrt{V} \tag{1.4}$$

so we can simply use the values we obtained for each lattice size and the standard deviation. We can average the various values for the constant α to give us an accurate answer. Averaging out these we get

$$\alpha \approx 0.2175$$

giving us our function,

$$\sigma = \frac{0.2175}{\sqrt{N_x \times N_y \times N_z \times N_T}}$$

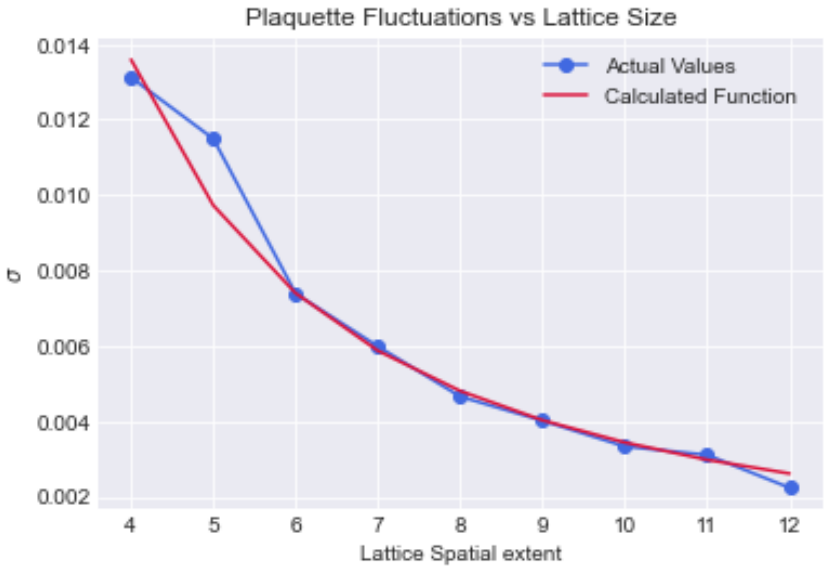

Figure 7: Plot of the standard deviation versus lattice size for $\beta = 1.9$ and lattice size running from $4^3 \times 4$ to $12^3 \times 4$. We plot this along side our calculated function $\frac{\alpha}{\sqrt{V}}$with the α we found from the average of all the standard deviations and using (1.4)

We see our function (1.4) that we created to model this behaviour fits the values very well, so with a good degree of certainty can say that the noise of the lattice scales with $\frac{\alpha}{\sqrt{V}}$. We can see the standard deviation (noise) decreasing as the lattice gets larger and our values for the average plaquette are more accurate as we go. This is what we expected as this is what happens as we get closer to the continuum limit, it more closely resembles the actual physics. I have displayed in the table below all the values for μ, σ, α obtained from the algorithm.

$\boldsymbol{\Lambda}$	$\boldsymbol{\mu}$	$\boldsymbol{\sigma}$	$\boldsymbol{\alpha}$
$4^3 \times 4$	0.4695	0.0131	0.2099
$5^3 \times 4$	0.4702	0.0115	0.2570
$6^3 \times 4$	0.4724	0.0074	0.2172
$7^3 \times 4$	0.4707	0.0060	0.2217
$8^3 \times 4$	0.4703	0.0047	0.2107
$9^3 \times 4$	0.4715	0.0040	0.2168
$10^3 \times 4$	0.4704	0.0033	0.2108
$11^3 \times 4$	0.4704	0.0031	0.2269
$12^3 \times 4$	0.4698	0.0022	0.1861
$16^3 \times 4$	0.4701	0.0019	0.2438

Table 1: Comparison of average plaquette values (μ), standard deviations (σ), and fit parameter (α) for different lattice sizes.

6.1.1 Correlation

Apparent in Figure 6, especially the $4^3 \times 4$ lattice is that the configurations tend to be 'correlated'. That is, subsequent runs have average plaquette values that lie quite close to each other. This is due to how the MC method selects new configurations and can negatively impact the accuracy of the measurements. The *auto-correlation* time is the amount of time it takes for each lattice site to become independent of each other. This is very important to keep track of. For observables that are computationally expensive to compute we want to try to reduce the correlation between subsequent measurements. One way to do this is by discarding a number of configurations between each measurement to help reduce correlation. Standard n_{cor} lie between 20 and 100 sweeps[11].

7 Integration Measure

Here I want to briefly introduce the integration measure for the link variables. Up to now we have only been considering a model with no quarks just gluons. This self-interaction between the gluons makes gluodynamics (QCD with no quarks) a very important theory to study. Pure gluodynamics exhibit color confinement, an integral part of QCD. This formalism is much easier to handle than QCD with quarks and we can use it as a springboard to full QCD by introducing quarks into our model.

The expectation value of observables O (which will be a function of the gluon fields) is given by

$$\langle O \rangle = \frac{1}{Z} \int \mathcal{D}U \; O(U) e^{-S[U]} = \frac{1}{Z} \int \mathcal{D}U \; O(U) e^{-\beta \sum_{x,\mu,\nu} (1 - P_{\mu\nu}(x))} \tag{1.5}$$

where the partition function Z is

$$Z = \int \mathcal{D}U X(U) e^{-S[U]}$$

The integration measure for the link variables is the product measure

$$\int \mathcal{D}U = \prod_{n \in \Lambda} \prod_{\mu=1}^{4} \int dU_\mu(n)$$

This denotes the integral over all possible values of all the link variables.

Here S denotes the action[7]

$$S[U] = \frac{\beta}{3} \sum_{n \in \Lambda} \sum_{\mu < \nu} Tr[I - U_{\mu\nu}(n)]$$

The β is the lattice spacing as before, which we will vary to see how it changes the system.

8 Algorithm

8.1 Simulation

Our aim is to use Monte Carlo integration to approximate integral (1.5), by generating configurations of link variables (our gluon fields) with probability $\propto \exp(-S[U])$. We could approach this the standard way using some kind of Metropolis algorithm. The only issue with this is that it is what we call a local algorithm. Such algorithms only allow for very small steps in the Markov chain, so we would have to perform many steps before we get an uncorrelated configuration. So we will circumvent this by using a non-local algorithm, where large subsets of field variables are changed in one go. The *coup de grâce* is the Heatbath algorithm which I explain below.

8.2 Staples

The first thing we want to note is that if we want to update only a single link $U_\mu(x)$, we need the part of the action $S[U]$ that depends on it. That is the plaquettes that contain the link $U_\mu(x)$. There are two plaquettes that contain the link variable. We already have one side of the square (plaquette), so we need the other 3 sides. Returning to equation (1.1) we see we need

$$U_\nu(x+\hat{\mu})U_\mu^\dagger(x+\hat{\nu})U_\nu^\dagger(x) \qquad U_\nu^\dagger(x+\hat{\mu}-\hat{\nu})U_\mu^\dagger(x-\hat{\nu})U_\nu(x-\hat{\nu})$$

These are called the 'staples' show below, it is easy to see where they get their name.

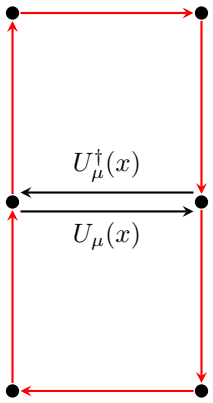

Figure 8: Staples shown in red, forming two elementary plaquettes with $U_\mu(x)$

We combine these into the *staple sum*

$$S_\mu(x) = \sum_{\nu\neq\mu} \left[U_\nu(x+\hat{\mu})U_\mu^\dagger(x+\hat{\nu})U_\nu^\dagger(x) + U_\nu^\dagger(x+\hat{\mu}-\hat{\nu})U_\mu^\dagger(x-\hat{\nu})U_\nu(x-\hat{\nu})\right]$$

We are finally ready to form the action for a given site x and direction $\hat{\mu}$

$$\begin{aligned} S[U_\mu(x)] &= S_0 - \frac{\beta}{2}\mathrm{Tr}(U_\mu(x)S_\mu(x)) \\ &= S_0 - \frac{\beta}{2}\mathrm{Tr}\ U_\mu(x)\sum_{\nu\neq\mu} U_\nu(x+\hat{\mu})U_\mu^\dagger(x+\hat{\nu})U_\nu^\dagger(x) \\ &\qquad + U_\nu^\dagger(x+\hat{\mu}-\hat{\nu})U_\mu^\dagger(x-\hat{\nu})U_\nu(x-\hat{\nu}) \end{aligned}$$

Up to now we have been representing our SU(2) matrices by 4 real numbers. Can we do the thing with the staple sum? No it turns out you can't as $S_\mu(x)$ is not an SU(2) matrix . But fortunately it is proportional to one, so we can write

$$S_\mu(x) = S_0 I + i\vec{\sigma}\cdot\vec{S} = \xi \qquad \xi = \sqrt{S_0^2 + \vec{S}^2} \tag{1.6}$$

where $s \in \mathrm{SU}(2)$ and (S_0, S_1, S_2, S_3) will be the four numbers representing the staple sum.

8.3 Proposing new links

We now have all the tools needed to perform the heatbath algorithm. The idea is to create SU(2) matrices $U'_\mu(x)$ for each x and μ, randomly distributed in accordance with

$$P(U'_\mu(x)) \propto \exp\left(\frac{\beta}{2}\,\mathrm{Tr}(U'_\mu(x)S_\mu(x))\right) \tag{1.7}$$

Using the representation explained in the previous section and the substitution

$$a = U'_\mu(x)s$$

where $a \in \mathrm{SU}(2)$,the problem then reduces to creating these random a matrices with the distribution

$$P(a) \propto \exp\left(\beta\xi\ \mathrm{Tr}\left(\frac{a}{2}\right)\right)$$

We do that in the following way:

1. We start by generating random SU(2) matrices for each link variable. This called a hot start, we could also have used a cold start where every link variable is set to the identity matrix, it is a matter of preference.

2. From this initial configuration we compute ξ and s from (1.6)

3. Now generate a random number a_4 with the distribution

$$P(a_4) \propto \sqrt{1-a_4^2}e^{\beta\xi a_4}$$

   we outline the procedure in it's own section below as it is the critical step.

4. Assuming we have got this a_4 we now want to generate three random numbers a_1, a_2, a_3 uniformly distributed on the surface of a sphere with radius $\sqrt{1-a_4^2}$. We do this by generating uniformly distributed random numbers on the surface. We can choose ϕ as a random number such that $\phi \in [0, 2\pi]$. We also need a $\theta \in [0, \pi]$. We need to be a

bit careful here with the spherical coordinates, as we need $\theta = 2\arccos(x)$ where x is a uniform random number in $[0, 1]$. From all this we can then get

$$a_1 = r\sin(\theta)cos(\phi) \quad a_2 = r\sin(\theta)sin(\phi) \quad a_3 = r\cos(\theta)$$

where $r = \sqrt{1 - a_4^2}$

5. We now have four numbers (a_1, a_2, a_3, a_4), we use these as our 4 real numbers to represent an SU(2) matrix, and we can form the matrix a. We use this as such

$$U'_\mu(x) = as^\dagger$$

This gives us our new link variable $U'_\mu(x)$ for site x and direction μ.

6. We repeat this process for every single x and μ until we have gone through the whole lattice and thus we have an entirely new configuration. This is one single sweep.

7. How many sweeps we perform after thermalisation depends on the accuracy needed and the computation time. Normally 300 or so sweeps is enough as this represents the system well enough, while not being too heavy on computation time.

8.3.1 Random Variable a_4

As promised I outline here how to obtain a_4 for the heatbath algorithm. The goal is to find values for a_4 distributed in accordance with $\sqrt{1 - a_4^2}e^{\alpha\beta a_4}$. We follow the procedure as explained in [2]. First we want to introduce a variable λ

$$a_4 = 1 - 2\lambda^2 \qquad a_4 \in [-1, 1]$$

We want to generate λ with the probability distribution

$$p1(\lambda) = \lambda^2 e^{-2\alpha\beta\lambda^2}$$

and we choose to accept or reject it based on

$$p_2(\lambda) = \sqrt{1 - \lambda^2}$$

The probability distribution p_1 is a Gaussian distribution modified by a polynomial. There are a few ways to random numbers with Gaussian distributions. We do it by

1. Starting with three random numbers r_1, r_2, r_3, uniformly distributed in $(0, 1]$. We have avoided choosing the number 0 here, since we later take the logarithm of these values. Since rng's usually cover the interval $[0, 1)$ we can just take $(1 - r_i)$ to do the same while avoiding zero.

$$\lambda^2 = -\frac{1}{2\alpha\beta}(\ln(r_1) + \cos^2(2\pi r_2)\ln(r_3))$$

This will follow the required distribution.

2. Now we need to incorporate the accept/reject step according to $p_2(\lambda)$, so we only accept values of λ which obey
$$r' \leq \sqrt{1-\lambda^2}$$
where r' is a random number uniformly distributed in $[0,1)$.

3. If we accept this λ this gives us our desired variable a_4 from
$$a_4 = 1 - 2\lambda^2$$

9 Overrelaxation

An alternative method of selecting new link variables is to choose the link variable that minimises the action. The Overrelaxation algorithm seeks to alter our lattice link variables as much as possible so as to move through the configuration space as much as possible.

This algorithm is much faster than both the Metropolis and heatbath.The only downside is that this never alters the action of the system. This called the microcanonical ensemble. Since we want to determine configurations according to the canonical ensemble, meaning it is distributed according to the Boltzmann weight, we must use it in conjunction with another algorithm, such as the heatbath algorithm we have used.

This algorithm works by utilising the property of the Metropolis algorithm that new configurations are always accepted if they don't alter the action. In the heatbath algorithm we start with a single link variable $U_\mu(x) = U$ with the probability distribution (1.7). While considering the fixed background of its neighbours. The sum of staples $S_\mu(x)$ is used in this context. The overrelaxation method finds a new value U' which has the same probability weight as U and so is automatically accepted.

The suggested change comes from the ansatz[7]
$$U \to U' = s^\dagger U_\mu^\dagger(x) s^\dagger$$

We see that
$$\mathrm{Tr}[U' S_\mu(x)] = \mathrm{Tr}[s^\dagger U_\mu^\dagger(x) s^\dagger \; S_\mu(x)] = \xi \; \mathrm{Tr}[s^\dagger U^\dagger] = \mathrm{Tr}[S_\mu^\dagger(x) U^\dagger] = \mathrm{Tr}[U S_\mu(x)]$$

This choice of U' does indeed leave the action invariant. On occasion we get that $\xi = 0$ so we just accept any random link variable. Implementation of this algorithm is efficient and useful for SU(2), however for other gauge groups such as $SU(3)$ it most likely isn't useful as it becomes much too complicated. Overrelaxation for SU(3) has been studied in [1] for those interested.

10 Polyakov Loop

So far we have been dealing with a model with no quarks, or at least one where the quarks have no effect on the gluons*. Which in many cases is a reasonable approximation. Now we want to try add a quark into our system and see what happens. First we need a very important quantity in the study of quarks the *Polyakov loop*.

It is also called a Wilson thermal loop and we'll see why. It is a modification of the Wilson loop where we focus on the time direction. We consider two paths that wrap around the entire time dimension. Due to our periodic boundary conditions we can't gauge-transform all the link variables to the identity I. We are able to however gauge the spatial components of the loop to I. So now our Wilson loop has reduced to two disconnected lines with opposite orientation,

$$T(m, N_T) \qquad T(n, N_T)^\dagger$$

They are made of link variables connecting points separated in time at fixed spatial points m and n. So they wind around our lattice in the time direction but in opposite directions.

This new observable can be made gauge invariant by taking the trace of the two loop separately, this is introduces our definition of the Polyakov loop P [10],

$$L_{\vec{x}} = \prod_{\tau=0}^{N_T-1} U_4(\vec{x}, \tau) \qquad P = \frac{1}{N_\sigma^4} \sum_{\vec{x}} \mathrm{Tr}(L_{\vec{x}})$$

The correlator $\langle P(\vec{x}_1)P(\vec{x}_2)^\dagger \rangle$ is related to the free energy F_q of a single, infinitely heavy quark in the following way

$$\langle P(\vec{x}_1)P(\vec{x}_2)^\dagger \rangle = e^{-\xi N_T F_q(\xi|\vec{x}_1 - \vec{x}_2|)} = e^{-F_q/T}$$

As per usual the energy has to be normalised at some distance, so at large distances i.e $\xi|\vec{x}_1 - \vec{x}_2| \to \infty$

$$\lim_{\xi|\vec{x}_1 - \vec{x}_2| \to \infty} \langle P(\vec{x}_1)P(\vec{x}_2)^\dagger \rangle = |\langle P \rangle|^2$$

So if $\langle P \rangle \neq 0$ we see that it only requires a finite amount of energy to add a single quark to the system, and thus in principle free quarks should be able to exist. Antipodally if the quarks are confined, i.e $F_q = \infty$ and $\langle P \rangle = 0$. In summary,

$$\langle P \rangle = 0 \qquad \text{Confinement}$$

$$\langle P \rangle \neq 0 \qquad \text{Deconfinement}$$

So there must exists some energy F_q and correspondingly a value of P, where the quarks transition from confinement to deconfinement. This is the critical temperature we discuss below.

We want to run through different values of β calculating the average Polyakov loop as we go, and try to find at what β this phase transition occurs at.

*Which would be the case if the quarks were infinitely heavy

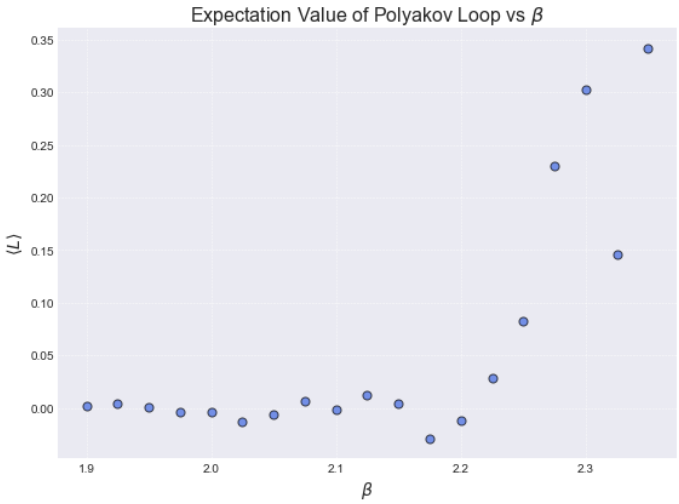

Figure 9: Polyakov loop expectation value demonstrating phase transition.

The above graph is the plot of the expectation value of the Polyakov loop $\langle P \rangle$ for β from $1.9 \to 2.4$. We can see for the first few values of β, the expectation value $\langle P \rangle$ is 0 up to some small error, this is the confined phase where the expectation value is zero. Then around $\beta = 2.2$ we see we get a sharp rise where $\langle P \rangle \neq 0$, this is the deconfinement phase we were after. The severity of the increase is related to the size of the lattice.

Consulting Table 2 we see when $\beta = 2.20 \implies T = 235\ MeV$ so our phase transition occurs around this temperature. Looking back at Figure 4 when we contrasted the average plaquette for different values of β, we concluded there was a possible phase transition around $\beta = 2.2$ which agrees exactly with our values from the Polyakov loop.

Another helpful quantity to help confirm our results is the Polyakov Loop susceptibility

$$\chi_L = \Lambda(\langle P^2 \rangle - \langle P \rangle^2)$$

This should peak at the phase transition as seen in the graph below

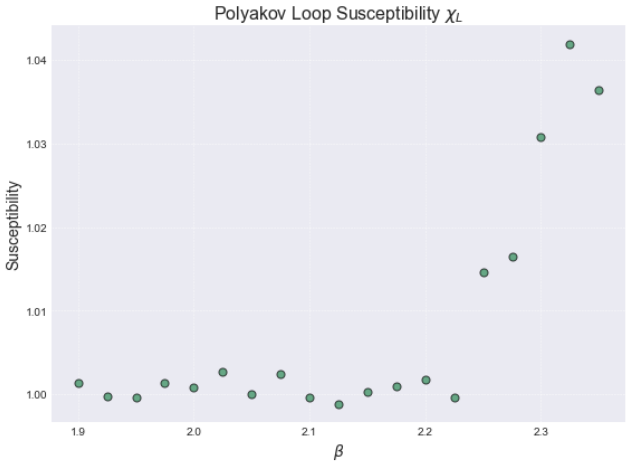

Figure 10: Polyakov Susceptibility

This graph is a bit less clear cut but we can still see that it has a peak around $\beta = 2.3$, which matches up with the expectation value of the Polyakov loop and indicates the phase transition around $\beta = 2.2$. The slight differences can be attributed to finite size effects and the fact that the Polyakov loop susceptibility is much more sensitive to fluctuations especially around the phase transition where there is a sharp rise in values. All this is excellent as all of our plots and measurements agree on the same value for this phase transition on our $8^3 \times 4$ lattice, so we can conclude relatively confidently that this is where the our phase transition occurs.

10.1 Critical Temperature

Now that we have the tools and can collect the expectation value of the Polyakov loop and the susceptibility we can calculate the critical temperature β_c above which the quarks will become deconfined. This will present itself in the expectation value of the Polyakov loop it should be 0 below β_c and then should shoot up after β_c to a non zero value. This will also be shown in the susceptibility χ_L as it will spike and have a peak at β_c. So using our results and table 1 we get

$$\beta = 2.20 \implies T = 235\ MeV$$

$$\beta = 2.30 \implies T = 298\ MeV$$

So our critical temperature lies somewhere in the range

$$235\ MeV \leq \beta_c \leq 298\ MeV$$

11 What is β?

Up until now I have been a bit vague on what exactly β is. I have declared it to be the lattice spacing and have alluded to it being representative of temperature. Now i will explain exactly how beta relates to the lattice spacing and temperature. β parametrises the strength of the strong interaction. In actuality the lattice spacing is defined by the value a. a and β are inversely proportional, so as we increase β the lattice spacing decreases. This relationship is very important since as we increase β the lattice sites get closer and closer together, thus becoming closer to the continuum limit and becoming more accurate. We need this lattice spacing to be small enough so it is representative of the physics but not so small that it becomes too computationally expensive to use.

β is defined in terms of the coupling constant g. This constant determines the strength of the force exerted in the interaction[4].

$$\beta = \frac{2n_{\text{color}}}{g_0^2}$$

where n_{color} is the number of colors, which in our case is 2. The actual relation between β and a is quite complicated and well beyond the scope of this project. One thing we do know is that it is an exponential relationship. Specifically a decays exponentially with increasing β.

11.1 Extrapolation

We can use this and some known values of β and a to obtain values of a for any given β. See the table below for the known values of β and a along with inverse a and the temperature. We will get to the latter two in a bit. If we want to obtain values of a in between known values we can use interpolation which will give us pretty accurate values. But this is not much good to us as we have started our lattice simulation from $\beta = 1.9$, so interpolation will do us no good here. Instead we want to use extrapolation to find values for a for these lower values of β.

We start by considering this exponential relationship between the two. We are going to assume that the relationship follows something of the form

$$a(\beta) = Ae^{-B\beta}$$

for some constants A and B, we can extrapolate the constants from our known data set and use these to obtain values of β outside our known range.

We could use exponential regression, taking the log of both sides and then fitting it with a linear model. But this is less accurate, luckily the scipy package has a function designed to fit non linear functions (exponential in our case) to data, so we can use this instead. This is straightforward enough and the graph we get below shows the values we get for a.

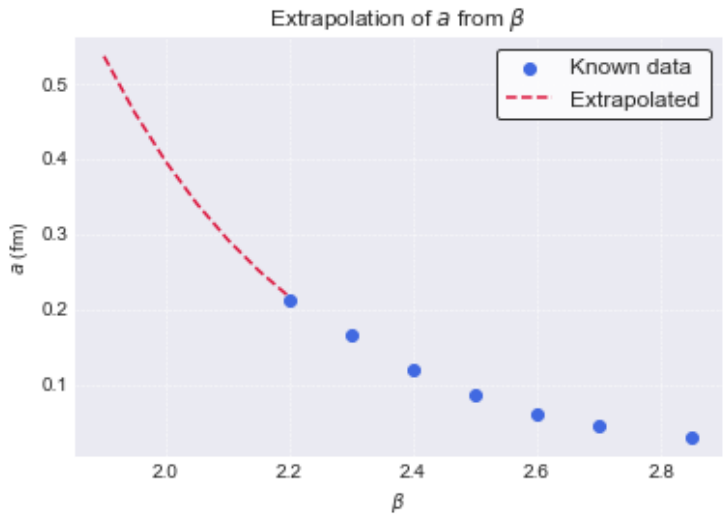

Figure 11: Extrapolation of a versus β from known values, exact values presented in 2

Now we do need to declare the caveat with this method. This whole calculation is predicated on the assumption that the relationship can be written as an exponential. When we are using extrapolation we need to be very careful as outside our known our the model could behave completely differently to what we expect. Any small uncertainties in the fitting parameters will be exacerbated the further we venture from our known values. Now we have stayed relatively close to our known values so we can be confident enough they will suffice. But without independent verification ranging further outside this interval is not advised. Since we don't actually use these values for a in our actual simulation, we run no risk. This more for an illustration of the relationship between β, a and T.

You may be wondering what physical interpretation a has and why is it given in fm or femtometers. $1\ fm = 1 \times 10^{-15}$ is a typical distance in the realm of protons, neutrons and quark systems. This makes it the natural choice of distance when we want to simulate quarks and gluons. As mentioned before a balance must be struck between choosing values of β and so values of a. A *coarse* lattice is usually defined[3] as having $a > 0.1$. So for this simulation we are indeed using a coarse lattice. In general coarse lattices are less computationally heavy but with the downside is that their relation to the continuum limit is much less accurate. Luckily we can use something called a tadpole improvement, enabling us to get pretty accurate results even on a coarse lattice. We detail the tadpole improvement in the next section.

11.2 Temperature

We also are interested in the relationship between β and the temperature T. Luckily this is a much more straightforward relationship given by

$$T = \frac{1}{N_\tau a}$$

where in our case $N_\tau = 4$. We can know concatenate all our results into a table of values as shown below.

β	a (fm)	a^{-1} (GeV)	T (MeV)
1.90	0.5358	0.368	92
1.95	0.4603	0.429	107
2.00	0.3955	0.499	125
2.05	0.3399	0.581	145
2.10	0.292	0.676	169
2.15	0.2509	0.786	197
2.20	0.211	0.938	235
2.30	0.166	1.192	298
2.40	0.120	1.654	414
2.50	0.0857	2.310	578
2.60	0.0612	3.235	809
2.70	0.0457	4.33	1083
2.85	0.02845	6.98	1745

Table 2: Lattice spacings temperatures for various values of β. Values above the horizontal line are extrapolated from our above algorithm; those below are from established data.

As we can see due to this inverse relation between β and a and our definition of the temperature, we get a nice relation between β and T. As we increase β, T also increases. This is what leads to the previously nebulous relationship between the two, which is now hopefully a bit clearer.

12 Uncertainty and Error Analysis

One important aspect of Monte Carlo simulations is the performing of error analysis of the observables. This tells us for example how many samples we should discard in between measuring observables to reduce the autocorrelation. We want to be able to assign some estimate to the statical error of our average values.

Computing the actual autocorrelation time is a complicated process and is often too computationally expensive in Monte Carlo simulations of QFT. There are however some simpler methods that still give us a good idea of the correlation of the data.

12.1 Data Blocking Methods

We divide our data up into sub-blocks of data of size K, we compute the mean values of the blocks and then treat these as new variables X_i. The variance of these new X_i values should decrease with $1/K$ if the original data is independent. We repeat this for various values of K until this $1/K$ decrease is observed then we know our data is independent of each other. Once we know that our data is independent we can then compute our observables and their errors. However, often the data sample is too small to get a reliable estimate of the variance. We can solve this by combining it with Jackknife or Bootstrap methods.[7]

12.2 Jackknife

The jackknife method is a cross-validation method. It is primarily used in estimating how accurately a predictive model will perform. The method is straightforward and gives us the standard deviation σ_θ.

We start with the data set of the observable we want to measure call it θ, say our data set has N elements. We compute the value of the observable from the entire data set and call this $\hat{\theta}$. We now systematically go through our data set and create N subsets of size $N-1$ by removing one value of the set at a time. We then calculate the observable with respect to these smaller data sets individually. There give us these θ_n for each set. Then the standard deviation is obtained by taking the square root of,

$$\sigma_{\hat{\theta}}^2 \equiv \frac{N-1}{N} \sum_{n=1}^{N} \left(\theta_n - \hat{\theta}\right)^2$$

For the final result we give $\langle\theta\rangle = \hat{\theta} \pm \sigma_{\hat{\theta}}$ or we can replace $\hat{\theta}$ with the unbiased estimator. Which is obtained from the bias

$$\tilde{\theta} \equiv \frac{1}{N} \sum_{n=1}^{N} \theta_n$$

which gives us $\hat{\theta} - (N-1)(\tilde{\theta} - \hat{\theta})$ as the unbiased estimator for $\langle\theta\rangle$

So now we want to get some values for our observables $P_{\mu\nu}, P, \chi_L$. First for the plaquette,

$$P_{\mu\nu} = 0.593264 \pm 0.002522 \qquad \text{Unbiased } : 0.593264$$

So we can see the accuracy of the average plaquette is quite high, I didn't bother separating the plaquette values per β, up into the confined and deconfined phase since we can

see from Figure 4 that it increases essentially linearly as we increase β with only a slight kink around the phase transition which will not skew our values by any significant amount. Also the plaquette is a local observable averaged over the spatial and time directions so it has a low variance.

Now for the Polyakov loop and susceptibility we will want to split it up into confined and deconfined phase since we have a sharp increase once we reach the critical temperature, this will throw off our mean values and can inflate our error. To demonstrate this we calculate the average value and error without splitting our values up,

$$\chi_L = 61.649128 \pm 28.483502 \qquad \langle L \rangle = 0.056614 \pm 0.038796$$

We can see that looking at Figure 9 we see this can't be right cause there are clearly values that aren't near zero. This is what we mean by it can affect our mean values, so let's split it up into the two phases.

Confined phase:

$$\langle L \rangle = -0.002552 \pm 0.003104 \qquad \chi_L = 0.217000 \pm 0.133285$$

Deconfined phase:

$$\langle L \rangle = 0.135501 \pm 0.086028 \qquad \chi_L = 121.254459 \pm 72.189650$$

We see the error for the Polyakov loop and the susceptibility are much higher as compared to the plaquette. This is due to a few factors,

- The Polyakov loop is a non-local observable averaged over the time direction and we only have one Polyakov loop per spatial site as opposed to the 4 of the plaquette so we have smaller sample size for the Polyakov loop.
- The susceptibility is by definition the variance so it is measuring the fluctuations which are quite high near the phase transition.
- Finally the Polyakov loop is a computationally expensive observable and has a high autocorrelation time again reducing the number of independent samples and increasing uncertainties.

So it is expected that the errors on the average value and susceptibility of the Polyakov loop are quite high.

13 Tadpole Improvement

Our entire model here is attempting to use discretised time and space (lattice) to approximate continuous spacetime. In the, what is called the continuum limit $a \to 0$, lattice gauge theories approximate this continuum very well. But the further we are from this continuum (on coarse lattices) limit the less accurate it is. This is due to errors and distortions that arise in any sort of numerical simulation dealing with the discretisation of spacetime. We call these distortions *lattice artifacts*. Some examples of lattice artifacts are

- Discretisation errors arising when a is far from 0.
- Finite volume effects can distort physical quantities if the volume is too small.
- Anisotropy, if spatial and temporal dimensions are different this can introduce an artificial direction dependence in the results.

Tadpole diagrams are induced by the non-linear connection between the lattice link variables $U_\mu(x)$ and the continuum gauge fields $A_\mu(x)$

$$U_\mu(x) \equiv e^{iagA_\mu(x)}$$

Luckily it turns out these effects can be mostly mitigated with a simple mean field renormalisation of the links

$$U_\mu(x) \to \frac{U_\mu(x)}{u_0}$$

We can determine this u_0 in a number of ways, it is still debated over which choice of operator is optimal to use. For isotropic lattices (same dimensions in every direction) we mostly use

$$u_0 \equiv \left\langle \frac{1}{N} \text{Re Tr}(U_{P_{\mu\nu}}) \right\rangle^{\frac{1}{4}}$$

where $U_{P_{\mu\nu}}$ is the average plaquette.

For simulations of the static quark potential however, it has been shown that using the mean Landau gauge

$$u_0 \equiv \left\langle \frac{1}{N} \text{Re Tr}(U_\mu) \right\rangle$$

reduces errors further[5].

The tadpole improvements rescale the link variables so they more closely resemble their continuous counterparts. This is especially useful when computing observables that directly depend on the link variables. In our case the Polyakov loop is susceptible to these lattice artifacts. We only apply the tadpole improvement once our algorithm has thermalised as doing so before hand can lead to unphysical results and further distortions.

14 Further Research and Improvements

14.1 Computational Efficiency

Computational time was the primary bottleneck in this project as is typical of lattice QCD. A balance has to be struck between choosing a fine enough lattice structure so that we get a system that is representative of the actual physics and but is not so computationally heavy that it takes too long to run. To run the $8^3 \times 4$ algorithm for $\beta = 1.9 \rightarrow 2.4$ with a burn in of 20 iterations and 1000 sweeps per β, took about 5 hours to run. Given the complexity and size of the data structure we are dealing with even for the $8^3 \times 4$ lattice there is only so much we can do. Due to the limited computational power of a standard laptop and as a result of the exponential increase in complexity as the model advanced, it made it unfeasible to simulate systems that approach the realistic parameters within the given time frame. That being said there are a few improvements that could be made to decrease the computational time.

A few small improvements could contribute to efficiency. As previously mentioned using pre-calculated index arrays which help implement nearest neighbour calculations at the same time as boundary conditions as these array calculations. Obtaining these indexes does take up a non trivial amount of time.

A common practice in Monte Carlo methods is to use an adaptive β, since β is what controls the acceptance rate of new configurations we can tweak the incremental change of β to target a specific acceptance rate. The heatbath method by design always returns a new configuration, but in the random variable step we keep choosing new values of a variable δ until it meets the given constraints, see 8.3.1. This δ depends on β so an adaptive step size can be used that tweaks β for a desired acceptance rate. For Monte Carlo methods the standard acceptance rate interval is $40\% - 60\%$. Another advantage to the adaptive step size is that it can reduce unnecessary computation in less critical areas. As shown in the plots in previous section, specifically the expectation value of the Polyakov loop, it is zero until we get to the phase transition so these areas aren't of as much interest to us as the area around the transition. So using a larger step size for values away from β_c then decreasing the step size as we approach the critical temperature can save computational time.

One obvious solution would be to just run the code on a computer with more power than ours. Since the code was written with readability and debugging in mind there are a lot of places where it is less than efficient. I chose python as the coding language since it's speed of development and wide range of libraries, made it the most effective given the limited time. But things such as dynamic typing and it's interpretive nature do create problems of efficiency that could be solved using faster languages such as C++, Julia or Matlab[12].

14.2 Parallel Programming

The main improvement would be to implement parallel programming techniques. A lot of calculations in lattice theories are what we call 'embarrassingly parallel' which means that there is minimal or no dependency upon communication between the parallel tasks, or for results between them. Evident especially when calculating the average plaquette, we only need the loop around four lattice sites for each plaquette, this is independent of the rest of it's neighbours. So we could slice up the lattice in a given way and calculate the plaquettes of each slice in parallel.

If one succeeds in efficiently distributing a task over n cores the theoretical minimum processing time is reduced by a factor of n. Of course we can never really reach this reduction since the actual distribution of tasks takes up computational time. This issue is quite large in Python and takes up a significant amount of computational time. But even with this parallelising updates and other observables on our lattice still decreases computational time significantly.

The implementation of such techniques is a non-trivial process. If one were to naively just divide up the lattice into chunks and update these at the same time we run the risk of updating links that are being used by another core. The size of the Wilson loops used in our algorithm impacts how we can divide up our lattice. Methods of dividing the lattice into pieces is called *masking*[11].

It is crucial that we identify what links of our lattice can be updated simultaneously. The action involves only 1×1 Wilson loops (Plaquettes) so whenever a given link is being updated we have to make sure we are not trying to update one of the links for a different plaquette that shares that link, similarly for the staples.

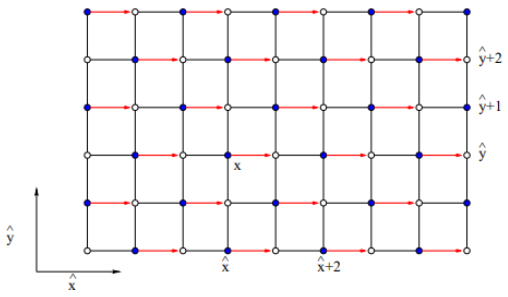

Figure 12: Checkerboard masking for the x-y plane of our lattice. The links in red can be updated simultaneously. [6]

Let us consider links in the $\hat{x}$ direction, we see from Figure 12 that we can choose a so called 'checkerboard' pattern of links that can be updated at the same time since they don't overlap with any staples or plaquettes that are to be updated. We can update exactly half of the links in the $\hat{x}$ direction at the same time [6]. This is the conventional standard for parallelising lattice updates for an action based on 1×1 Wilson loops.

14.3 Extend to SU(3)

In actuality every quark comes in 3 colors and the gluons are described by 3×3 unitary matrices. Our simulation only dealt with 2 colors and 2×2 unitary matrices. This acts as a building block in moving to SU(3) allowing us to write our algorithm in a simpler setting and then theoretically it should not be two much trouble adapting our algorithm to deal with $SU(3)$ matrices. The main difference here being that instead of representing our matrices using four real number as shown in 4.0.1, we need to use 18 real coefficients (9 complex numbers).

One possible way of proposing new link configurations for SU(3) is as follows, the candidate link variable is given by

$$U'_\mu(x) = XU_\mu(n)$$

where the matrix X can be constructed using the same methods as outlined in 8.3 but then embedding these SU(2) matrices in 3×3 matrices,

$$R = \begin{pmatrix} r_{11} & r_{12} & 0 \\ r_{21} & r_{22} & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad S = \begin{pmatrix} s_{11} & 0 & s_{12} \\ 0 & 1 & 0 \\ s_{21} & 0 & s_{22} \end{pmatrix} \quad T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & t_{11} & t_{12} \\ 0 & t_{21} & t_{22} \end{pmatrix}$$

and this give us a possible choice for X

$$X = RST$$

Since the matrices R, S, T are close to the identity matrix, so will X be. To preserve the symmetry of our selection probability we would need to choose X and $X^\dagger$ with equal probability[7]. So we could build up a set of these X matrices using the above definition for X and randomly select a matrix from this set. Then we would reset this set every now and then to avoid reselecting the same matrices.

On top of this the next logical step would be to increase the lattice size to perhaps $16^3 \times 4$ which would give us a better approximation and accurate results that more closely replicate the behaviour of the continuous model.

15 Dynamical Fermions

Our model up to now has been using what is called a 'quenched' approximation. Quenched quantum chromodynamics is an approximation used often in lattice QCD where the effects of dynamical quarks are ignored, so we are solely looking at gluon dynamics and interactions. This approximation is used because in Monte Carlo simulations it is fairly straightforward to model interacting gluons. The introduction of dynamical fermions (quarks) would be the natural next step. Calculations with these dynamical fermions are much more challenging than the quenched calculations and require new methods and algorithms, such as Hybrid Monte Carlo algorithms and Grassmann numbers[7]. I would have liked to introduce these fermions into my model to achieve more comprehensive and accurate results, but the limited time frame of this project did not permit it. However, their inclusion presents an intriguing avenue for future research.

16 Conclusion

In this project, I have constructed and validated a lattice-based simulation of the strong force using SU(2) gauge theory, focusing on the confinement and deconfinement phases of quarks. Through use of Monte Carlo and heatbath algorithms, and by analysing observables, namely the average plaquette and Polyakov loop, the simulation successfully reproduced known QCD behaviour. This includes a clear confinement–deconfinement phase transition at the critical temperature around $\beta_c = 2.2$ The results align well with theoretical predictions and established numerical findings. This confirms the accuracy and reliability of the simulation.

Although the model employs SU(2) rather than the more realistic SU(3), it captures the essential physics of QCD and provides a solid foundation for extending the simulation to full SU(3) and dynamical fermions. The computational techniques developed here, including error reduction and analysis, heatbath, overrelaxation, and tadpole improvement, demonstrate the potential of efficient lattice methods even on coarse lattices. Implementing the above suggestions for further optimisation and the introduction of parallelisation, this framework could be scaled to address more complex systems and tackle meaningful problems in lattice QCD.

17 Appendix

Below is the complete Python code used to produce the results and graphs presented in this paper. The code is organized in the same order as the results appear in the paper, with references to the corresponding figures where applicable. It includes comments where necessary for clarity, along with a brief description explaining its purpose and implementation details

17.1 Average Plaquette versus β

The below code for the average plaquette against β was ran for values of $1.0 \rightarrow 3.0$ for β on a $4^3 \times 4$ lattice with 100 sweeps per β. It produced plot 4.

```
"""
This code analyses the average plaquette at different values of β ranging
    from 1.0 to 3.0 with a step size of 0.1
"""

import numpy as np
from numpy import log,pi,arccos,sin, cos
import matplotlib.pyplot as plt
from datetime import datetime
import statistics


plt.style.use('seaborn-v0_8-darkgrid')
start_time = datetime.now()

print('Start Time:', start_time.strftime('%H:%M:%S'))


betas = np.arange(1.0,3.0,0.1)
nwarm = 20
runs = 100


directions = np.array([[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])


# This is the Main_function tweaked to be applied to any size of lattice, we
    explain what each function does in the Main_Function file
def initial_kick(Nx,Ny,Nz,Nt):
    U = np.zeros((Nx, Ny, Nz, Nt, 4, 4))
    for i in range(Nx):
        for j in range(Ny):
            for k in range(Nz):
                for l in range(Nt):
                    for mu in range(4):
                        U[i, j, k, l, mu] = random_vars()
    return U


def trace(u):
    return 2*u[0]

```

```
41
42
43 def nn(u,v):
44     w0 = u[0]*v[0]-u[3]*v[3]-u[1]*v[1]-u[2]*v[2]
45     w1 = u[0]*v[1]+u[3]*v[2]+u[1]*v[0]-u[2]*v[3]
46     w2 = u[0]*v[2]-u[3]*v[1]+u[1]*v[3]+u[2]*v[0]
47     w3 = u[0]*v[3]+u[3]*v[0]-u[1]*v[2]+u[2]*v[1]
48     return np.array([w0,w1,w2,w3])
49
50 def nd(u,v):
51     w0 = u[0]*v[0]+u[3]*v[3]+u[1]*v[1]+u[2]*v[2]
52     w1 = -u[0]*v[1]-u[3]*v[2]+u[1]*v[0]+u[2]*v[3]
53     w2 = -u[0]*v[2]+u[3]*v[1]-u[1]*v[3]+u[2]*v[0]
54     w3 = -u[0]*v[3]+u[3]*v[0]+u[1]*v[2]-u[2]*v[1]
55     return np.array([w0,w1,w2,w3])
56
57 def dn(u,v):
58     w0 = u[0]*v[0]+u[3]*v[3]+u[1]*v[1]+u[2]*v[2]
59     w1 = u[0]*v[1]-u[3]*v[2]-u[1]*v[0]+u[2]*v[3]
60     w2 = u[0]*v[2]+u[3]*v[1]-u[1]*v[3]-u[2]*v[0]
61     w3 = u[0]*v[3]-u[3]*v[0]+u[1]*v[2]-u[2]*v[1]
62     return np.array([w0,w1,w2,w3])
63
64 def dd(u,v):
65     w0 = u[0]*v[0]-u[3]*v[3]-u[1]*v[1]-u[2]*v[2]
66     w1 = -u[0]*v[1]+u[3]*v[2]-u[1]*v[0]-u[2]*v[3]
67     w2 = -u[0]*v[2]-u[3]*v[1]+u[1]*v[3]-u[2]*v[0]
68     w3 = -u[0]*v[3]-u[3]*v[0]-u[1]*v[2]+u[2]*v[1]
69     return np.array([w0,w1,w2,w3])
70
71 def random_vars():
72     u = np.random.randn(4)
73     norm = np.linalg.norm(u)
74     return u/norm
75
76
77 def staple(x,mu,U,Nx,Ny,Nz,Nt):
78     staple_sum = np.array([0,0,0,0])
79     lattice = np.array([Nx,Ny,Nz,Nt])
80     for v in range(4):
81         if v != mu:
82             xmu = np.mod((x + directions[mu]),lattice)
83             x_plus_v  = np.mod((x + directions[v]), lattice)
84             xmuv = np.mod((x + directions[mu] - directions[v]),lattice)
85             x_minus_v = np.mod((x - directions[v]),lattice)
86
87             first = (nn(U[tuple(xmu)+ (v,)],dd(U[tuple(x_plus_v)+ (mu,)],U[
    tuple(x)+ (v,)])))
88             second = dn(U[tuple(xmuv)+ (v,)],dn(U[tuple(x_minus_v)+ (mu,)],U[
    tuple(x_minus_v)+ (v,)]))
89             staple_sum = staple_sum + first + second
90     return staple_sum
91
92 def action(x,mu,U):
93     stap = staple(x,mu,U)
94     return stap[0] - beta/2 * trace(nn(U[tuple(x)+ (mu,)],stap))
95
```

```
def plaquette(x,mu,v,U,Nx,Ny,Nz,Nt):
    lattice = np.array([Nx,Ny,Nz,Nt])
    xmu = np.mod((x + directions[mu]),lattice)
    xv = np.mod((x + directions[v]), lattice)

    first = dd(U[tuple(xv)+ (mu,)],U[tuple(x)+ (v,)])
    second = nn(U[tuple(xmu)+ (v,)],first)
    third = nn(U[tuple(x)+ (mu,)],second)
    return 1/2 * trace(third)

def average_plaquette(U,Nx,Ny,Nz,Nt):
    plaq_sum = 0
    for i in range(Nx):
        for j in range(Ny):
            for k in range(Nz):
                for l in range(Nt):
                    for mu in range(4):
                        for v in range(mu+1,4):
                            plaq_sum += plaquette([i,j,k,l], mu, v,U,Nx,Ny,Nz
    ,Nt)

    return 1/(6*(Nx**3*Nt)) * plaq_sum




def randfn(a):
    R = 2
    d = 0
    count = 0
    while R**2 > 1 - d/2:
        X1 = -np.log(np.random.rand())/a
        X2 = -np.log(np.random.rand())/a
        C = np.cos(2 * np.pi * np.random.rand())**2
        d = X1 * C + X2
        R = np.random.rand()
        count += 1
    return 1 - d,count


def heatbath(U,beta,Nx,Ny,Nz,Nt):
    count = []
    for i in range(Nx):
        for j in range(Ny):
            for k in range(Nz):
                for l in range(Nt):
                    for mu in range(4):
                        x = np.array([i, j, k, l])
                        stap = staple(x,mu,U,Nx,Ny,Nz,Nt)
                        xi = norm(stap)
                        s = stap/xi
                        a4 ,c= randfn(beta*xi)
                        count.append(c)
                        r = np.sqrt(1-a4**2)

                        # Generate random angles
```

```
                        th = np.pi * np.random.rand()
                        phi = 2 * np.pi * np.random.rand()

                        cth = np.cos(th)
                        sth = np.sqrt(1 - cth * cth)

                        a = [a4,r * sth * np.cos(phi), r * sth * np.sin(phi),
     r * cth]
                        g_new = nd(a, s)
                        U[i,j,k,l,mu] = g_new
    return U

def norm(m):
    return np.sqrt(m[0]**2+m[1]**2+m[2]**2+m[3]**2)




plaquette_vals4 = []
plaq_betas = []

# Here we just run through the values of beta fora 4^3 x 4 lattic calculating
    the average plaquette at each beta.
for b in range(len(betas)):
    U4 = initial_kick(4,4,4,4)
    plaquette_vals4 = []

    for _ in range(nwarm):
        U4 = heatbath(U4,betas[b],4,4,4,4)

    for a in np.arange(1,runs):
        U4 = heatbath(U4,betas[b],4,4,4,4)
        plaquette_vals4.append(average_plaquette(U4,4,4,4,4))

    plaq_betas.append(np.mean(plaquette_vals4))


plt.scatter(betas, plaq_betas, color='royalblue', label=r'⟨P⟩',s=70)
plt.title(r'Average Plaquette vs β')
plt.xlabel(r'β')
plt.ylabel('Plaquette')
plt.grid(True)
plt.show()

end_time = datetime.now()
print('Duration: {}'.format(end_time - start_time))
```

Listing 1: Calculation of the average plaquette for various values of β

17.2 Average Plaquette Thermalisation

This algorithm computes how long the burn-in period is for various lattice sizes using the plaquette operator. We can see from6 which this produced, it takes about 20 sweeps to thermalise, which the algorithm then oscillates about a constant value for the plaquette about 0.47.

```
"""
This code analyses the burn-in period and convergence of the average
    plaquettte to a roughly constant value for various lattice sizes

"""
import numpy as np
from numpy import log,pi,arccos,sin, cos
import matplotlib.pyplot as plt
from datetime import datetime
import Main_Function

plt.style.use('seaborn-v0_8-darkgrid')
bleu = (52/255, 207/255, 235/255)
greeen = (165/255, 235/255, 52/255)

start_time = datetime.now()

print('Start Time:', start_time.strftime('%H:%M:%S'))

beta = 1.9
runs = 200

plaquette_vals4 = []
plaquette_vals8 = []
plaquette_vals12 = []
plaquette_vals16 = []
action_vals = []


U4 = initial_kick(4,4,4,4)
U12 = initial_kick(12,12,12,4)
U16 = initial_kick(16,16,16,4)

# For the 8^3 x 4 matrix we load in a precalculated array
loaded_flat_matrix = np.loadtxt("matrix.txt")
U8 = loaded_flat_matrix.reshape(8, 8, 8, 4, 4, 4)

# Calculating average plaquette values and updating link variables using the
    heatbath algorithm
for a in np.arange(1,runs):

    U4 = heatbath(U4,beta,4,4,4,4)
    U8 = heatbath(U8,beta,8, 8, 8, 4)
    U12 = heatbath(U12,beta,12, 12, 12, 4)
    U16 = heatbath(U16,beta,16, 16, 16, 4)

    plaquette_vals4.append(average_plaquette(U4,4,4,4,4))
    plaquette_vals8.append(average_plaquette(U8,8,8,8,4))
    plaquette_vals12.append(average_plaquette(U12,12,12,12,4))
    plaquette_vals16.append(average_plaquette(U16,16,16,16,4))
```

```
plt.plot(plaquette_vals4, label=r'$4^3 \times 4$ lattice', color='royalblue')
plt.plot(plaquette_vals8, label=r'$8^3 \times 4$ lattice', color='crimson')
plt.title('Average Plaquette Values vs Configurations')
plt.xlabel('Configurations')
plt.ylabel('Average Plaquette')
plt.grid(True)
plt.legend()
plt.show()


end_time = datetime.now()
print('Duration: {}'.format(end_time - start_time))
```

Listing 2: Analysis of thermalisation period of average plaquette for various lattice sizes.

17.3 Noise Analysis

```
"""
This code analyses the noise (standard deviation) of the average plaquette
    for lattice sizes from 4^3 x 4 to 12^3 x 4. It calculates the standard
    deviation and how much it oscillates about the constant value of the
    average plaquette that is independent of lattice size. We also compose a
    function that calculates how the noise decreases as lattice size increases
     for any size lattice.
"""

import numpy as np
import Main_function
from numpy import log,pi,arccos,sin, cos
import matplotlib.pyplot as plt
from datetime import datetime
import statistics

plt.style.use('seaborn-v0_8-darkgrid')
bleu = (52/255, 207/255, 235/255)
greeen = (165/255, 235/255, 52/255)

start_time = datetime.now()
print('Start Time:', start_time.strftime('%H:%M:%S'))

beta = 1.9
nwarm = 20
runs = 100
constant = []
deviation = []

lattice = np.arange(4,13)

# Runs the algorithm for various lattice sizes and calulates the mean value
    of the plaquette aswell as the standard deviation

for i in range(4,13):
    U = initial_kick(i,i,i,4)
```

```
    plaquette_vals = []

    for _ in range(nwarm):
        U= heatbath(U,beta,i,i,i,4)

    for a in np.arange(1,runs):
        U = heatbath(U,beta,i,i,i,4)
        plaquette_vals.append(average_plaquette(U,i,i,i,4))

    constant.append(np.mean(plaquette_vals))
    deviation.append(np.std(plaquette_vals))


# Then we want to calculate the constant alpha for our fitted function
    allowing us to formulate a function for the noise of a given lattice

alpha_vals = []
for i in range(len(deviation)):
    N = i+4
    alpha_vals.append(deviation[i] * np.sqrt(N**3 * 4))

print("Alpha: ", np.mean(alpha_vals))
expected = []

for i in lattice:
    expected.append(np.mean(alpha_vals)/np.sqrt(i**3*4))

plt.figure()
plt.plot(lattice,deviation, label='Actual Values',color='royalblue')
plt.plot(lattice,expected,label='Calculated Function ',color='crimson')
plt.xlabel('Lattice Spatial extent')
plt.ylabel(r'$\sigma$')
plt.legend()
plt.grid(True)
plt.title('Plaquette Fluctuations vs Lattice Size')
plt.show()



end_time = datetime.now()
print('Duration: {}'.format(end_time - start_time))
```

Listing 3: Analysis of Monte Carlo noise for various lattice sizes using the average plaquette. Code for plot 7

17.4 Extrapolation of β versus a

```
"""
The code below extrapolates from known values of β and the corresponding a
    values new a values for lower values of β.

"""

import numpy as np
import scipy.optimize as scp
import matplotlib.pyplot as plt

plt.style.use('seaborn-v0_8-darkgrid')

# We use the given values of beta and the corresponding values of a
beta = np.array([2.20, 2.30, 2.40, 2.50, 2.60, 2.70, 2.85])
a = np.array([0.211, 0.166, 0.120, 0.0857, 0.0612, 0.0457, 0.02845])


def model(beta, A, B):
    return A * np.exp(-B * beta)

# Using curve fitting from scipy with an initial guess of (1,1), giving us
    the constants A and B
parameters, covaraiance = scp.curve_fit(model, beta, a, p0 = (1.0,1.0))
A = parameters[0]
B = parameters[1]

beta_guess = np.arange(1.9, 2.2, 0.05)
a_guess = model(beta_guess, A, B)

for i in range(len(beta_guess)):
    print(f"Beta = {beta_guess[i]}, a = {a_guess[i].6f}")

plt.scatter(beta, a, label='Known data',color='royalblue')
plt.plot(beta_guess, a_guess,'--', label='Extrapolated', color='crimson')
plt.title(r'Extrapolation of a from β')
plt.xlabel(r'β')
plt.ylabel(r'a (fm)')
plt.grid(True)
plt.legend()
plt.show()
```

Listing 4: Extrapolation of lattice spacing values a for various β. Prodcued plot 11

17.5 Main algorithm for Polyakov Loops

The following code forms the core of the project, serving as the foundation for all observables, graphs, and measurements. It generates Figures 9 and 10.

```
"""
Here is the primary algorithm upon which all other code and functions depends. Here we perform the heatbath algorithm and overrelaxation method for a range of β and sweeps. We calculate and store the average plaquette, Polyakov loop expectation and susceptibility and plot these values.
"""

import numpy as np
from numpy import log,pi,arccos,sin, cos
import matplotlib.pyplot as plt
from datetime import datetime

start_time = datetime.now()

print('Start Time:', start_time.strftime('%H:%M:%S'))
plt.style.use('seaborn-v0_8-darkgrid')

# Define lattice dimensions
Nx = 8
Ny = 8
Nz = 8
Nt = 4
betas = np.arange(2.0,2.7,0.05)
lattice = np.array([Nx,Ny,Nz,Nt])
nwarm = 20 # Number of sweeps for thermalisation
runs = 100
discard = 1 # How many configurations we discard between measurements


directions = np.array([[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])


# Here the initial kick is a hot start that gives each link variable a random configuration

def random_vars():
    u = np.random.randn(4)
    norm = np.linalg.norm(u)
    return u/norm

def initial_kick():
    U = np.zeros((Nx, Ny, Nz, Nt, 4, 4), dtype=float)
    for i in range(Nx):
        for j in range(Ny):
            for k in range(Nz):
                for l in range(Nt):
                    for mu in range(4):
                        U[i, j, k, l, mu] = random_vars() # Returns a random normalised vector representing the four real numbers in the matrix representation
    return U

```

```
47
48 # Takes the parameters and returns the matrix in terms of the pauli basis
49 def makematrix(u):
50     return np.array([[u[0] + u[3]*1j, u[1]*1j+u[2]],[u[1]*1j-u[2],u[0]-u[3]*1
       j]])
51
52 def trace(u):
53     return 2*u[0]
54
55 # Here we define how to multiply any two matrices in our chosen
       representation, n represents a normal matrix while d is the daggered
       version of the matrix.
56 def nn(u,v):
57     w0 = u[0]*v[0]-u[3]*v[3]-u[1]*v[1]-u[2]*v[2]
58     w1 = u[0]*v[1]+u[3]*v[2]+u[1]*v[0]-u[2]*v[3]
59     w2 = u[0]*v[2]-u[3]*v[1]+u[1]*v[3]+u[2]*v[0]
60     w3 = u[0]*v[3]+u[3]*v[0]-u[1]*v[2]+u[2]*v[1]
61     return np.array([w0,w1,w2,w3])
62
63 def nd(u,v):
64     w0 = u[0]*v[0]+u[3]*v[3]+u[1]*v[1]+u[2]*v[2]
65     w1 = -u[0]*v[1]-u[3]*v[2]+u[1]*v[0]+u[2]*v[3]
66     w2 = -u[0]*v[2]+u[3]*v[1]-u[1]*v[3]+u[2]*v[0]
67     w3 = -u[0]*v[3]+u[3]*v[0]+u[1]*v[2]-u[2]*v[1]
68     return np.array([w0,w1,w2,w3])
69
70 def dn(u,v):
71     w0 = u[0]*v[0]+u[3]*v[3]+u[1]*v[1]+u[2]*v[2]
72     w1 = u[0]*v[1]-u[3]*v[2]-u[1]*v[0]+u[2]*v[3]
73     w2 = u[0]*v[2]+u[3]*v[1]-u[1]*v[3]-u[2]*v[0]
74     w3 = u[0]*v[3]-u[3]*v[0]+u[1]*v[2]-u[2]*v[1]
75     return np.array([w0,w1,w2,w3])
76
77 def dd(u,v):
78     w0 = u[0]*v[0]-u[3]*v[3]-u[1]*v[1]-u[2]*v[2] # changed this line
79     w1 = -u[0]*v[1]+u[3]*v[2]-u[1]*v[0]-u[2]*v[3]
80     w2 = -u[0]*v[2]-u[3]*v[1]+u[1]*v[3]-u[2]*v[0]
81     w3 = -u[0]*v[3]-u[3]*v[0]-u[1]*v[2]+u[2]*v[1]
82     return np.array([w0,w1,w2,w3])
83
84
85
86 # Returns the staple sum S_μ(x) for a given lattice site x and direction μ
       while taking boundary conditions into account
87 def staple(x,mu,U):
88     staple_sum = np.array([0,0,0,0])
89     for v in range(4):
90         if v != mu:
91             xmu = np.mod((x + directions[mu]),lattice)
92             x_plus_v  = np.mod((x + directions[v]), lattice)
93             xmuv = np.mod((x + directions[mu] - directions[v]),lattice)
94             x_minus_v = np.mod((x - directions[v]),lattice)
95
96             first = (nn(U[tuple(xmu)+ (v,)],dd(U[tuple(x_plus_v)+ (mu,)],U[
       tuple(x)+ (v,)])))
97             second = dn(U[tuple(xmuv)+ (v,)],dn(U[tuple(x_minus_v)+ (mu,)],U[
       tuple(x_minus_v)+ (v,)]))
```

```
            staple_sum = staple_sum + first + second
    return staple_sum

def action(x,mu,U):
    stap = staple(x,mu,U)
    return stap[0] - beta/2 * trace(nn(U[tuple(x)+ (mu,)],stap))

def plaquette(x,mu,v,U):
    xmu = np.mod((x + directions[mu]),lattice)
    xv = np.mod((x + directions[v]), lattice)

    first = dd(U[tuple(xv)+ (mu,)],U[tuple(x)+ (v,)])
    second = nn(U[tuple(xmu)+ (v,)],first)
    third = nn(U[tuple(x)+ (mu,)],second)
    return 1/2 * trace(third)

def average_plaquette(U):
    plaq_sum = 0
    for i in range(Nx):
        for j in range(Ny):
            for k in range(Nz):
                for l in range(Nt):
                    for mu in range(4):
                        for v in range(mu+1,4):
                            plaq_sum += plaquette([i,j,k,l], mu, v,U)

    return 1/(6*(Nx**3*Nt)) * plaq_sum


def norm(m):
    return np.sqrt(m[0]**2+m[1]**2+m[2]**2+m[3]**2)

# Returns the value a_4 for use in heatbath when obtaining a new link
    configurations
def randfn(a):
    R = 2
    d = 0
    count = 0
    while R**2 > 1 - d/2:
        X1 = -np.log(np.random.rand())/a
        X2 = -np.log(np.random.rand())/a
        C = np.cos(2 * np.pi * np.random.rand())**2
        d = X1 * C + X2
        R = np.random.rand()
        count += 1
        a_4 = 1-d
    return a_4,count


def heatbath(U,beta):
    count = []
    for i in range(Nx):
        for j in range(Ny):
            for k in range(Nz):
                for l in range(Nt):
                    for mu in range(4):
                        x = np.array([i, j, k, l])
```

```
                        stap = staple(x,mu,U)
                        xi = norm(stap)
                        s = stap/xi
                        a4 ,c= randfn(beta*xi)
                        count.append(c)
                        r = np.sqrt(1-a4**2)

                        # Generate random angles
                        th = np.pi * np.random.rand()
                        phi = 2 * np.pi * np.random.rand()

                        cth = np.cos(th)
                        sth = np.sqrt(1 - cth * cth)

                        a = [a4,r * sth * np.cos(phi), r * sth * np.sin(phi),
     r * cth]
                        g_new = nd(a, s)
                        U[i,j,k,l,mu] = g_new

    return U

# Overrelaxation algorithm to be run 5 times after each heatbath update to
    help explore the sample space more efficiently
def overrelaxation(U):
    for i in range(Nx):
        for j in range(Ny):
            for k in range(Nz):
                for l in range(Nt):
                    for mu in range(4):
                        x = np.array([i, j, k, l])
                        stap = staple(x,mu,U)
                        s = stap/norm(stap)
                        s = [s[0],-s[1],-s[2],-s[3]]
                        w = dd(s,U[x[0],x[1],x[2],x[3],mu])
                        g_new = nd(w,s)
                        U[i, j, k, l, mu] = g_new

    return U

# Function to test that direction around plaquette is irrelevant, useful for
    checking the plaquette function is working correctly
def test_plaquette(x,mu,v):
    ting = initial_kick()
    clockwise = plaquette(x, mu, v,ting)
    counter = plaquette(x, v, mu, ting)
    print(clockwise)
    print(counter)


# Calculation of Polyakov loop, we compute L and L^2 for the susceptibility
def polyakov_loop(U):
    total_p = 0
    total_p2 = 0
    spatial = Nx * Ny * Nz

    for i in range(Nx):
        for j in range(Ny):
```

```
            for k in range(Nz):
                pl = U[i, j, k, 0, 3]

                for t in np.arange(1, Nt):
                    pl = nn(pl, U[i, j, k, t, 3])

                p = trace(pl)
                total_p += p
                total_p2 += (p**2)

    avg_p = total_p/spatial
    avg_p2 = total_p2/spatial

    return avg_p, avg_p2

# Tadpole improvement to be implemented after thermalisation to improve approximation to continuous model
def tadpole(U):
    u_0 = (average_plaquette(U))**0.25
    U_new = U/u_0
    return U_new


L_avg = []
L2_avg = []
plaq_betas = []

# Here we apply the heatbath and overrelaxation methods looping over all values of beta

for b in range(len(betas)):
    L_vals = []
    L_2vals =[]
    U = initial_kick() # Give all the link variables an intial radnom configurations
    plaquette_vals = []

    # Thermalisation
    for _ in range(nwarm):
        U= heatbath(U,betas[b])

    U = tadpole(U) # Apply tadpole improvement after thermalisation

    for a in range(runs):
        U = heatbath(U,betas[b])

        for _ in range(5):
           U = overrelaxation(U)

        L,L2 = polyakov_loop(U)
        L_vals.append(L)
        L_2vals.append(L2)
        plaquette_vals.append(average_plaquette(U))


    plaq_betas.append(np.mean(plaquette_vals))
    L_avg.append(np.mean(L_vals))
```

```
    L2_avg.append(np.mean(L_2vals))
    print(f"Beta:{betas[b]}")


lattice_volume = Nx * Ny * Nz * Nt
L_avg = np.array(L_avg)
L2_avg = np.array(L2_avg)
var = lattice_volume*(L2_avg - L_avg**2)


# Expectation value of Polyakov loop
plt.scatter(betas, L_avg, color='crimson', label=r'⟨L⟩', s=50,)
plt.title(r'Expectation Value of Polyakov Loop vs β')
plt.xlabel(r'β')
plt.ylabel(r'⟨L⟩')
plt.grid(True)
plt.show()

# Susceptibility
plt.scatter(betas, var, color='green', label='Susceptibility', s=50)
plt.title(r'Polyakov Loop Susceptibility $\chi_L$')
plt.xlabel(r'β')
plt.ylabel('Susceptibility')
plt.grid(True)
plt.show()

end_time = datetime.now()
print('Duration: {}'.format(end_time - start_time))
```

Listing 5: Main function for computing Polyakov Loop and susceptibility

17.6 Jackknife for Polyakov Loop

```
"""
This performs a Jackknife analysis of the Polyakoov loop expectatio nvalue
    and suscpetibility.
It makes use of values obtained from the main algorithm and saved for
    analysis.
I have split up the values between the confined and deconfined phase to give
    an accurate representation of mean values and errors as the phase
    transition
 introduces a sharp increase that can skew the average values if not dealt
    with correctly. It also calculates the unbiased operator which gives a
    mean value of the observable accounting for the error.
"""

import numpy as np
import matplotlib.pyplot as plt
import statistics

# Take in saved values for the Polyakov loop
poly = np.array(np.loadtxt('L_avg_1.9_2.3_0.025.txt'))
suscep = np.array(np.loadtxt('L2_avg_1.9_2.3_0.025.txt'))

poly2 = np.array(np.loadtxt('L_avg_2.3_2.4_0.025.txt'))
suscep2 = np.array(np.loadtxt('L2_avg_2.3_2.4_0.025.txt'))
```

```
18
19
20 loop = np.concatenate((poly, poly2))
21 susceptibility = np.concatenate((suscep, suscep2))
22
23 betas = np.arange(1.9, 2.35, 0.025)
24 index = np.where(np.isclose(betas, 2.20))[0] # get index of phase transiton
25 volume = 8 * 8 * 8 * 4
26
27 def jackknife(L):
28
29     # Split into confined and deconfined phases
30     L1 = L[:12]
31     L2 = L[12:]
32
33     # Calculate overall mean
34     L_mean_con = np.mean(L1)
35     L_mean_decon = np.mean(L2)
36
37     L2_mean_con = np.mean(L1**2)
38     L2_mean_decon = np.mean(L2**2)
39
40     chi_con = volume * (L2_mean_con -L_mean_con**2)
41     chi_decon = volume * (L2_mean_decon - L_mean_decon**2)
42
43     std_vals_chi_con = 0
44     std_vals_chi_decon = 0
45
46     std_vals_L_con = 0
47     std_vals_L_decon = 0
48     N1 = len(L1)
49     N2 = len(L2)
50
51     bias_chi_con = 0
52     bias_L_con = 0
53
54     bias_chi_decon = 0
55     bias_L_decon = 0
56
57     # Errors for confined phase
58     for n in range(N1):
59         theta_L = np.delete(L1,n)
60         theta_L_mean = np.mean(theta_L)
61         theta_L2_mean = np.mean(theta_L**2)
62         chi_theta = (volume * (theta_L2_mean - theta_L_mean**2))
63         std_vals_chi_con += (chi_theta-chi_con)**2
64         std_vals_L_con += (theta_L_mean - L_mean_con)**2
65
66         bias_chi_con += chi_theta
67         bias_L_con += theta_L_mean
68
69     # Errors for deconfined phase
70     for n in range(N2):
71         theta_L = np.delete(L2,n)
72         theta_L_mean = np.mean(theta_L)
73         theta_L2_mean = np.mean(theta_L**2)
74         chi_theta = (volume * (theta_L2_mean - theta_L_mean**2))
```

```
        std_vals_chi_decon += (chi_theta-chi_decon)**2
        std_vals_L_decon += (theta_L_mean - L_mean_decon)**2

        bias_chi_decon += chi_theta
        bias_L_decon += theta_L_mean

    chi_thetacon = np.sqrt((N1-1)/N1 * std_vals_chi_con)
    chi_thetadecon = np.sqrt((N2-1)/N2 * std_vals_chi_decon)

    L_con = np.sqrt((N1-1)/N1 * std_vals_L_con)
    L_decon = np.sqrt((N2-1)/N2 * std_vals_L_decon)

    # Unbiased operator calculations
    chi_con_tilde =  1/N1 * bias_chi_con
    L_con_tilde =  1/N1 * bias_L_con

    chi_decon_tilde =  1/N2 * bias_chi_decon
    L_decon_tilde =  1/N2 * bias_L_decon

    unbiased_chi_con = chi_con - (N1-1) * (chi_con_tilde- chi_con)
    unbiased_L_con = L_mean_con - (N1-1) * (L_con_tilde- L_mean_con)

    unbiased_chi_decon = chi_decon - (N2-1) * (chi_decon_tilde- chi_decon)
    unbiased_L_decon = L_mean_decon - (N2-1) * (L_decon_tilde- L_mean_decon)

    return chi_thetacon, L_con, chi_thetadecon, L_decon, chi_con, chi_decon,
    L_mean_con,L_mean_decon, unbiased_chi_con,unbiased_chi_decon,
    unbiased_L_con,unbiased_L_decon


chi_con_err, L_con_err, chi_decon_err, L_decon_err, chi_con, chi_decon,
    L_mean_con, L_mean_decon, unbiased_chi_con,unbiased_chi_decon,
    unbiased_L_con,unbiased_L_decon = jackknife(loop)

print("Confined phase:")
print(f"L = {L_mean_con} + {L_con_err}, chi= {chi_con} + {chi_con_err}")


print("Deconfined phase:")
print(f"L = {L_mean_decon} + {L_decon_err}, chi = {chi_decon} + {
    chi_decon_err}")

print("Confined phase unbiased:")
print(f"  L = {unbiased_L_con}   chi = {unbiased_chi_con}")
print("Deconfined phase unbiased:")
print(f"  L = {unbiased_L_decon}  chi = {unbiased_chi_decon}")
```

Listing 6: Jackknife analysis of Polyakov loop expectation value and susceptibility and produces results in the Jackknife section.

17.7 Jackknife for Average Plaquette

```python
import numpy as np
import matplotlib.pyplot as plt
import statistics

plaqs = (np.loadtxt('plaq_betas.txt'))
N = len(plaqs)
volume = 8 * 8 * 8 * 4

def jackknife(plaq):
    plaq_mean = np.mean(plaq)
    std_vals_plaq = 0
    bias = 0

    for n in range(N):
        theta_plaq = np.delete(plaq,n)
        theta_plaq_mean = np.mean(theta_plaq)
        std_vals_plaq += (theta_plaq_mean -plaq_mean)**2
        bias += theta_plaq_mean

    theta_tilde =  1/N * bias
    unbiased = plaq_mean - (N-1)*(theta_tilde - plaq_mean)

    return np.sqrt((N-1)/N * std_vals_plaq) , plaq_mean, unbiased

plaq_error, plaq_mean, unbiased = jackknife(plaqs)

print(f" plaquette : {plaq_mean} + {plaq_error}")
print(f"Unbiased : {unbiased}")
```

Listing 7: Jackknife analysis for average plaquettes and produces results in the Jackknife section.

References

[1] J. Polonyi and K. Szlachanyi. *Phase transition from strong-coupling expansion*. 1982. DOI: https://doi.org/10.1016/0370-2693(82)91280-1. URL: https://www.sciencedirect.com/science/article/pii/0370269382912801.

[2] B. Svetitsky and L.G. Yaffe. *Nuclear Physics*. 1982.

[3] W. Dimm M. Alford and G.P. Lepage. *QCD on coarse Lattices*. 1994. DOI: https://doi.org/10.48550/arXiv.hep-lat/9412035.

[4] T. R. Klassen R. G. Edwards U. M. Heller. *The Effectiveness of Non-Perturbative O(a) Improvement in Lattice QCD*. 1997. DOI: https://doi.org/10.48550/arXiv.hep-lat/9711052.

[5] Norman H. Shakespeare and Howard D. Trottier. *Tadpole-improved SU(2) lattice gauge theory*. 1998. DOI: https://doi.org/10.48550/arXiv.hep-lat/9803024.

[6] Anthony Williams Frederic Bonnet Derek Leinweber. *General Algorithm For Improved Lattice Actions on Parallel Computing Architectures*. 2001. URL: https://doi.org/10.48550/arXiv.hep-lat/0001017.

[7] Christof Gattringer and Christian B.Lang. *Quantum Chromodynamics on a lattice*. 2010.

[8] Biagio Lucini Kurt Langfeld and Antonio Rago. *The density of states in Gauge Theories*. 2012. DOI: https://doi.org/10.48550/arXiv.1204.3243.

[9] David Tong. *Gauge Theory*. 2018. URL: https://www.damtp.cam.ac.uk/user/tong/gaugetheory/gt.pdf.

[10] David A. Clarke et al. *Polyakov Loop Susceptibility and Correlators in the Chiral Limit*. 2019. DOI: https://doi.org/10.22323/1.363.0194.

[11] Piter Annema. *Determining the String Tension using QCD Lattice Simulations*. 2019.

[12] Tim Daly. *Hybrid Monte Carlo Simulation of the Discretised Bosonic BFSS Model*. 2024. URL: https://www.dias.ie/wp-content/uploads/2024/09/Tim_Daly.pdf.

[13] *Color Charge and Confinement*. URL: https://fafnir.phyast.pitt.edu/particles/color.html.