Simulation of SU(3) Yang-Mills Theory on the Lattice

Author: Siddhartha Harmalkar

March 17, 2019

Contents

1 In the continuum

Quantum chromodynamics is the study of the dynamics of two interacting fields which transform non-trivially and differently under local SU(3) transformations:

$$\mathcal{L}_{\text{QCD}}(g, m_q) = \underbrace{-\frac{1}{2g^2}\text{Tr}[F_{\mu\nu}F^{\mu\nu}]}_{\equiv\ \mathcal{L}_G} + \underbrace{\sum_{q\in\{u,d,s,c,b,t\}} \bar{q}(\gamma^\mu\partial_\mu + m_q)q}_{\equiv\ \mathcal{L}_F} + \underbrace{\sum_{q\in\{u,d,s,c,b,t\}} \bar{q}(\gamma^\mu A_\mu)q}_{\equiv\ \mathcal{L}_{\text{int}}} \tag{1}$$

Which is a function of the bare coupling g and bare "quark masses" m_q. The full lagrangian of QCD can be thought of as consisting of three physical processes:

1. $\mathcal{L}_G$ describes free fields $A_\mu(x)$ which transform as elements of the adjoint representation of the local SU(3) transformation at each point given by $\Omega(x)$:

$$U_\mu^{(\Omega)}(x) = \Omega(x)U_\mu(x)\Omega(x)^{-1} \tag{2}$$

   This is also the Lagrangian of *SU(3) Yang-Mills* theory.[1]

2. $\mathcal{L}_F$ desrcibes free quarks $q(x)$ and anti-quarks $\bar{q}(x)$ which transform as elements of the fundamental and anti-fundamental representations of the transformation:

$$q^{(\Omega)}(x) = \Omega(x)q(x),\ \bar{q}^{(\Omega)}(x) = \bar{q}(x)\Omega^{-1}(x) \tag{3}$$

3. $\mathcal{L}_{\text{int}}$ ensures that the full lagrangian is invariant under Ω, and in doing so introduces interactions between the $A_\mu(x)$ and $q(x)$ fields. We introduced these interactions as the minimal prescription needed to ensure gauge invariance of a generalization of the local U(1) symmetry which the electron and photon fields transform under in QED to a theory of N electron-like (quarks) and N^2-1 photon-like (gluons) fields transforming under a local U(N) symmetry. We can also get to the same physics from the perspective of starting out with non-abelian Yang-Mills theory, which will be the focus of this lecture. The interactions in $\mathcal{L}_{\text{int}}$ can be viewed in a different way from this perspective: As the minimal prescription of a gauge-invariant lagrangian which adds sources that transform in the fundamental representation to the theory of non-abelian, non-interacting fields described by $\mathcal{L}_G$.

It will be beneficial to recall the construction of L_G and L_{int}, which arose because gauging the non-local symmetry $q(x) \to \Omega q(x)$ of $\mathcal{L}_F$ to a local one of the form $q(x) \to \Omega(x)q(x)$ required a non-trivial modification to the derivative term in $\mathcal{L}_F$, which is no longer gauge invariant:

$$\begin{aligned}\bar{q}(x)\partial_\mu q(x) \neq \bar{q}(x)^{(\Omega)}(x)\partial_\mu\psi^{(\Omega)}(x) &= \bar{q}(x)\Omega^{-1}(x)\partial_\mu\left(\Omega(x)q(x)\right)\\ &= \bar{q}(x)\Omega^{-1}(x)(\partial_\mu\Omega(x)q(x) + \Omega(x)\partial_\mu q(x))\end{aligned}$$

[1] "Yang-Mills theories" refer to quantized non-abelian gauge theories in general. See [3] for a good introduction to their history and importance.

Unless $\partial_\mu \Omega(x) = 0$, which is not true in general but of course holds true for global transformations. Our fix was a gauge "covariant" version of the ordinary derivative in direction n which acts as

$$n^\mu \partial_\mu|_x : q \to \lim_{\epsilon \to 0} \frac{1}{\epsilon}(q(x + \epsilon n) - q(x)) \tag{4}$$

that instead acts as

$$n^\mu D_\mu|_x : q \to \lim_{\epsilon \to 0} \frac{1}{\epsilon}(W(x, x + \epsilon n)q(x + \epsilon n) - q(x)) \tag{5}$$

Where $W(x, y)$ has the transformation property

$$W^{(\Omega)}(x, y) = \Omega(x)W(x, y)\Omega(y)^{-1} \tag{6}$$

and therefore solves our issue. Here is where we introduced the fields $A_\mu(x)$ as arbitrary vector fields which transform as

$$A_\mu^{(\Omega)}(x) = \Omega(x)A_\mu(x)\Omega(x)^{-1} + i(\partial_\mu \Omega(x))\Omega(x)^{-1} \tag{7}$$

and are the building blocks of an object, the *Wilson line*:

$$W(x, y) = \mathrm{P} \exp\left\{ ig \oint_y^x dz^\mu A_\mu(z) \right\} \ (P = \text{Path ordered}) \tag{8}$$

Which transforms as desired given the transformation rule for the gauge fields, and gives us an explicit form of the *covariant derivative*

$$D_\mu q(x) = \partial_\mu q(x) + igA_\mu(x)q(x) \tag{9}$$

Since:

$$W(x, x + \epsilon n) \approx 1 - ig\epsilon n^\mu A_\mu(x) \tag{10}$$

This "minimal subtraction" method of ensuring $\mathcal{L}_F$ is gauge invariant with the replacement $\partial_\mu \to D_\mu = \partial_\mu - igA_\mu$ is what led to the addition of the $\mathcal{L}_{\text{int}}$ term. We then constructed a kinetic term for the A_μ fields which allowed them to propagate, and since terms like $\partial_\mu A \partial^\mu A$ are not gauge invariant we were constrained to $\mathcal{L}_G$.

2 On the lattice

The path integral formalism allows us to discretize spacetime in a very natural way by introducing a lattice Λ of $|\Lambda|$ points in D-dimensional spacetime with dimensions L_i, separated by lattice spacing a which the fields that enter our Lagrangian will now be restricted to:[2]

[2]The lattice spacing is not evident in (11) because it does not affect the labeling of the points on the lattice. Instead, it will show up in the finite differences that take place of derivative terms in the continuum.

$$\Lambda = \{n = (n_1, n_2, \cdots, n_D) \mid 1 \leq n_i \leq L_i\},\ |\Lambda| \equiv \sum_{i=1}^{D} L_i \tag{11}$$

and then simply defining our integration measures to be of the form:

$$D\psi = \prod_{n\in\Lambda} d\psi(n),\ D\bar{\psi} = \prod_{n\in\Lambda} d\bar{\psi}(n) \tag{12}$$

for the fermions, and

$$DA = \prod_{\substack{n\in\Lambda \\ \mu\in\{1,\cdots,D\}}} dA_\mu(n) \tag{13}$$

for the gauge fields.

However, instead of the A_μ fields we will find it more convenient to work with the *link variables*

$$U_\mu(n) = \exp(-igaA_\mu(n)), \tag{14}$$

which are the lattice discretization of the Wilson line (8), since in the continuum,

$$W(n, n+\hat{\mu}) = 1 - igaA_\mu(x) + \mathcal{O}(a^2) = \exp(-igaA_\mu(n)). \tag{15}$$

Where $\hat{\mu}_\nu$ is the unit vector in direction μ scaled by the lattice spacing a, and takes us from lattice site $(n_1, \cdots, n_\mu, \cdots n_D)$ which is embedded in the continuum to the lattice site $(n_1, \cdots, n_\mu + 1, \cdots, n_D)$. Our measures of integration will be the lattice measures $D\psi$ and $D\bar{\psi}$ for the fermions as in (12), and a lattice measure DU for the links:

$$DU = \prod_{\substack{n\in\Lambda \\ \mu\in\{1,\cdots,D\}}} dU_\mu(n) \tag{16}$$

In order to check if our results to agree with the high-energy phenomena which we observe at the LHC, for example, we will need to take a *continuum limit*:

$$\begin{aligned}
D\psi^{(\text{cont})} &= \lim_{\substack{a\to 0 \\ |\Lambda|\to\infty}} c_1(a) \prod_{n\in\Lambda} d\psi(n) \\
D\bar{\psi}^{(\text{cont})} &= \lim_{\substack{a\to 0 \\ |\Lambda|\to\infty}} c_2(a) \prod_{n\in\Lambda} d\bar{\psi}(n) \\
DU^{(\text{cont})} &= \lim_{\substack{a\to 0 \\ |\Lambda|\to\infty}} c_3(a) \prod_{\substack{n\in\Lambda \\ \mu\in\{1,\cdots,D\}}} dU_\mu(n)
\end{aligned}$$

This is in fact a rigorous way to define the measures in the continuum expression

$$Z_{QCD}^{(\text{cont})} = \int D\psi^{(\text{cont})} D\bar{\psi}^{(\text{cont})} DU^{(\text{cont})} \exp(iS_{QCD}(\psi, \bar{\psi}, U)) \tag{17}$$

The lattice spacing plays the role of the ultraviolet regulator, rendering the quantum field theory finite. The continuum theory is recovered by taking the limit of vanishing lattice spacing, which can be reached by tuning the bare coupling constant to zero according to the renormalization group. Taking this limit in practice is a highly non-trivial task and will not be discussed here. Throughout the rest of these notes, we will consider lattices at finite (and fixed) lattice spacing a and volume $|\Lambda|$.

2.1 Describing LQCD observables in terms of the link variables

First let's define the following quantity:

$$\langle A(x) \rangle_{x \sim p(x)} \equiv \frac{\int p(x) A(x)}{\int p(x)} \tag{18}$$

This looks like the definition of the expected value of $A(x)$ when x is sampled according to the distribution $p(x)$. However, we will *not* be interpreting it as such at the moment, because this interpretation requires that $p(x)$ satisfies $p(x) \in R^{\geq 0}\ \forall x$, which is in general not going to be true for the functions which we identify as $p(x)$ in the formal manipulations below[3].

We can rewrite observables of QCD like so using this definition:

[3]For example, in a theory with only one quark flavor, $\int D\psi \mathcal{D}\bar{\psi}\ e^{-S_F(\psi,\bar{\psi},U)}$ can take on negative values for some gauge configurations U. In fact, it can even take on complex values for a theory of quarks at a nonzero chemical potential. There are still ways to calculate quantities of the form (18) with statistical methods even if $\exists x \mid p(x) \notin \mathbb{R}^{\geq 0}$. For example, one can decompose $p(x)$ as $p(x) = r(x)e^{i\theta(x)}$, where $r(x) = |p(x)| \in \mathbb{R}^{\geq 0}\ \forall x$, and then compute:

$$\langle \mathcal{O}(x) \rangle_{x \sim p(x)} = \frac{\int dx\ \mathcal{O}(x)p(x)}{\int dx\ p(x)} = \frac{\int dx\ \mathcal{O}(x)r(x)e^{i\theta(x)}}{\int dx\ r(x)e^{i\theta(x)}} = \frac{\int dx\ \mathcal{O}(x)r(x)e^{i\theta(x)} / \int dx\ r(x)}{\int dx\ r(x)e^{i\theta(x)} / \int dx\ r(x)} = \frac{\langle \mathcal{O}(x)e^{i\theta(x)} \rangle_{x \sim r(x)}}{\langle e^{i\theta(x)} \rangle_{x \sim r(x)}}$$

However, if $|\ \langle e^{i\theta(x)} \rangle_{x \sim r(x)}\ | \ll 1$, a large number of independent samples is required from $r(x)$ in order to separate the signal from the noise in the denominator, because a set of complex numbers which each have modulus 1 can only have an average modulus much smaller than unity due to significant cancellations coming from the different orientations of the numbers in the set, so in a statistical calculation of their average modulus we must sample these cancellations to get close to the true value. Without enough samples, the noise will dominate this ratio as a whole, since it appears in the denominator. This is the *sign*(al to noise) *problem* in finite density QCD [4]. Note that the fluctuations in $\theta(x)$ are what cause the numerical problem - if $\theta(x) = \theta_0\ \forall x$, then we would only need one sample, and there would in fact be no sign problem at all as $|\ \langle e^{i\theta_0} \rangle_{x \sim r(x)}\ | = 1$, but if $\theta(x) = |x|\ \forall x$, then we would never be able to sample enough points given an infinite region of support of $r(x)$ and this would be evident in the horrible sign problem $|\ \langle e^{i|x|} \rangle_{x \sim r(x)}\ | \ll 1$.

$$
\begin{aligned}
\langle \mathcal{O}(\psi,\bar{\psi},U)\rangle_{(\psi,\bar{\psi},U)\sim e^{-S(\psi,\bar{\psi},U)}} &= \frac{\int \mathcal{D}U\mathcal{D}\psi\mathcal{D}\bar{\psi}\ e^{-S(\psi,\bar{\psi},U)}\mathcal{O}(\psi,\bar{\psi},U)}{\int \mathcal{D}U\mathcal{D}\psi\mathcal{D}\bar{\psi}\ e^{-S(\psi,\bar{\psi},U)}} \\
&= \frac{\int \mathcal{D}U\mathcal{D}\psi\mathcal{D}\bar{\psi}\ e^{-S_G(U)-S_F(\psi,\bar{\psi},U)}\mathcal{O}(\psi,\bar{\psi},U)}{\int \mathcal{D}U\mathcal{D}\psi\mathcal{D}\bar{\psi}\ e^{-S_G(U)-S_F(\psi,\bar{\psi},U)}} \\
&= \frac{\int \mathcal{D}U\ e^{-S_G(U)}\int \mathcal{D}\psi\mathcal{D}\bar{\psi}\ e^{-S_F(\psi,\bar{\psi},U)}\mathcal{O}(\psi,\bar{\psi},U)}{\int \mathcal{D}U\ e^{-S_G(U)}\int D\psi\mathcal{D}\bar{\psi}\ e^{-S_F(\psi,\bar{\psi},U)}} \\
&= \frac{\int \mathcal{D}U\ e^{-S_G(U)}\left(\int D\eta\mathcal{D}\bar{\eta}\ e^{-S_F(\eta,\bar{\eta},U)}\right)\left(\frac{\int \mathcal{D}\psi\mathcal{D}\bar{\psi}\ e^{-S_F(\psi,\bar{\psi},U)}\mathcal{O}(\psi,\bar{\psi},U)}{\int D\eta\mathcal{D}\bar{\eta}\ e^{-S_F(\eta,\bar{\eta},U)}}\right)}{\int \mathcal{D}U\ e^{-S_G(U)}\int D\psi\mathcal{D}\bar{\psi}\ e^{-S_F(\psi,\bar{\psi},U)}} \\
&= \frac{\int \mathcal{D}U\ e^{-S_G(U)}\int D\eta\mathcal{D}\bar{\eta}\ e^{-S_F(\eta,\bar{\eta},U)}\left\langle \mathcal{O}(\psi,\bar{\psi},U)\right\rangle_{(\psi,\bar{\psi})\sim e^{-S_F(\psi,\bar{\psi},U)}}}{\int \mathcal{D}U\ e^{-S_G(U)}\int D\psi\mathcal{D}\bar{\psi}\ e^{-S_F(\psi,\bar{\psi},U)}} \\
&= \left\langle \left\langle \mathcal{O}(\psi,\bar{\psi},U)\right\rangle_{(\psi,\bar{\psi})\sim e^{-S_F(\psi,\bar{\psi},U)}}\right\rangle_{U\sim e^{-S_G(U)}\int D\eta\mathcal{D}\bar{\eta}\ e^{-S_F(\eta,\bar{\eta},U)}}
\end{aligned}
$$

In more compact notation:

$$
\langle \mathcal{O}\rangle = \left\langle \langle \mathcal{O}\rangle_F(U)\right\rangle_G \tag{19}
$$

Where

$$
\langle \mathcal{O}\rangle_F(U) = \frac{\int \mathcal{D}\psi\mathcal{D}\bar{\psi}\ e^{-S_F(\psi,\bar{\psi},U)}\mathcal{O}(\psi,\bar{\psi},U)}{\int D\psi\mathcal{D}\bar{\psi}\ e^{-S_F(\psi,\bar{\psi},U)}} \tag{20}
$$

and

$$
\langle \mathcal{O}\rangle_G = \frac{\int \mathcal{D}U\ e^{-S_G(U)}\mathcal{O}(U)\int D\psi\mathcal{D}\bar{\psi}\ e^{-S_F(\psi,\bar{\psi},U)}}{\int \mathcal{D}U\ e^{-S_G(U)}\int D\psi\mathcal{D}\bar{\psi}\ e^{-S_F(\psi,\bar{\psi},U)}} \tag{21}
$$

The study of "pure gauge theory" or "gluons" refers to the calculation of observables of the following form:

$$
\langle \mathcal{O}\rangle = \frac{\int \mathcal{D}U e^{-S_G(U)}\mathcal{O}(U)}{\int \mathcal{D}U e^{-S_G(U)}} \equiv \frac{1}{Z_G}\int \mathcal{D}U e^{-S_G(U)}\mathcal{O}(U) \tag{22}
$$

That is, the computation of observables $\mathcal{O}(U)$ which depend only on the link variables U with respect to the partition function Z_G.

2.2 Ensuring Gauge invariance of observables

In the continuum, QCD is a locally gauge-invariant theory. So on the lattice, we want expectation values of observables to be invariant under gauge transformations generated by local transformations $\Omega(x)$. In other words, we want the following condition to be satisfied:

$$\langle \mathcal{O} \rangle = \frac{1}{Z_G} \int \mathcal{D}U e^{-S_G(U)} \mathcal{O}(U) = \frac{1}{Z_G} \int \mathcal{D}U^{(\Omega)} e^{-S_G(U^{(\Omega)})} \mathcal{O}(U) \tag{23}$$

Where the gauge configuration $U^{(\Omega)}$ is related to U via $U_\mu^{(\Omega)}(x) = \Omega(x) U_\mu(x) \Omega(x+\hat{\mu})$. This can be accomplished by constructing a path integral measure which satisfies

$$\mathcal{D}U = \mathcal{D}U^{(\Omega)} \tag{24}$$

and an action which satisfies:

$$e^{-S_G(U^\Omega)} = e^{-S_G} \tag{25}$$

2.2.1 Constructing a gauge invariant integration measure

Remember that on a D-dimensional lattice Λ with sites labeled by $n \in \Lambda$ and directions $\mu \in \{1, 2, \cdots, D\}$, our path integration measure is simply:

$$\mathcal{D}U = \prod_{n,\mu} dU(n)_\mu. \tag{26}$$

A gauge transformation $\Omega(n)$ on the lattice has the effect of transforming the measure like so:

$$\mathcal{D}U = \prod_{n,\mu} dU(n)_\mu \to \mathcal{D}U^{(\Omega)} = \prod_{n,\mu} dU^{(\Omega)}(n)_\mu = \prod_{n,\mu} d\left(\Omega(n) U_\mu(n) \Omega(n+\hat{\mu})^{-1}\right) \tag{27}$$

Since $\Omega(n)$ and $\Omega(n+\hat{\mu})^{-1}$ are independent, we need a gauge invariant integration measure dU to satisfy:

$$\int_{\mathrm{SU}(3)} dU f(U) = \int_{\mathrm{SU}(3)} dU f(VUW) \ \forall V, W \tag{28}$$

We also want the integration to be normalized in some sense. A set of sufficient (but not necessary) conditions which ensures that the measure has the desired properties is:

$$\int_{\mathrm{SU}(3)} dU = 1, \int_{\mathrm{SU}(3)} dU f(U) = \int_{\mathrm{SU}(3)} d(VU) f(U) = \int_{\mathrm{SU}(3)} d(UV) f(U) \tag{29}$$

for arbitrary $V \in \mathrm{SU}(3)$. At the end of the day, we really only know how to integrate with respect to real numbers, so in order to construct such a measure we naturally start by parametrizing elements of the group by real parameters, and defining our measure over those parameters. Let $U(\omega)$ represent any element in $\mathrm{SU}(N)$ generated using N^2-1 real parameters ω:

$$U(\omega) = \exp\left(-i \sum_{\alpha=1}^{N^2-1} \omega^{(\alpha)} \lambda_\alpha\right) \tag{30}$$

Where λ_α are N^2-1 linearly independent elements in $\mathfrak{su}(N)$. In terms of these parameters, we are searching for an $f(\omega)$ such that

$$f(\omega)\prod_k d\omega^{(k)} = f(\tilde{\omega})\prod_k d\tilde{\omega}^{(k)} \tag{31}$$

Where $\tilde{\omega}$ are the parameters which generate $U(\omega)V$ or $VU(\omega)$. We know that

$$\prod_k d\omega^{(k)} = \det J \prod_k d\tilde{\omega}^{(k)} \tag{32}$$

where

$$J_{nm} = \frac{\partial\tilde{\omega}^{(n)}}{\partial\omega^{(m)}} \tag{33}$$

So we need $f(\omega)$ to satisfy

$$f(\omega)\det J = f(\tilde{\omega}) \tag{34}$$

We do this by building an object $g(\omega)$ which transforms as $g(\tilde{\omega}) = J^T g(\omega) J$ and therefore satisfies $\det g(\tilde{\omega}) = \det g(\omega)\det J^2$, which will allow us to construct $f(\omega)$ as:

$$f(\omega) = c\sqrt{\det g(\omega)} \tag{35}$$

Which satisfies the right transformation property:

$$f(\omega)\det J = c\sqrt{\det g(\omega)}\det J = c\sqrt{\frac{\det g(\tilde{\omega})}{\det J^2}}\det J = c\det g(\tilde{\omega}) = f(\tilde{\omega}) \tag{36}$$

An object which transforms under $\omega\to\tilde{\omega}$ as $g(\omega)$ should is:

$$g(\omega)_{nm} = \mathrm{Tr}\left(\frac{\partial U(\omega)}{\partial\omega^{(n)}}\frac{\partial U(\omega)^\dagger}{\partial\omega^{(m)}}\right) \tag{37}$$

Because, using the chain rule:

$$g(\tilde{\omega})_{nm} = \mathrm{Tr}\left(\frac{\partial U(\tilde{\omega})}{\partial\tilde{\omega}^{(n)}}\frac{\partial U(\tilde{\omega})^\dagger}{\partial\tilde{\omega}^{(m)}}\right) = \mathrm{Tr}\left(\frac{\partial U(\omega)}{\partial\omega^{(n)}}\frac{\partial U(\omega)^\dagger}{\partial\omega^{(m)}}\right)\times\underbrace{\left(\frac{\partial\omega^{(k)}}{\partial\tilde{\omega}^{(n)}}\frac{\partial\omega^{(l)}}{\partial\tilde{\omega}^{(m)}}\right)}_{J_{kn}J_{lm}=(J^T)_{nk}J_{lm}} = (J^T)_{nk}g(\omega)_{kl}J_{lm} \tag{38}$$

So we can construct a gauge invariant measure dU given by:

$$dU = c\sqrt{\det g(\omega)}\prod_k d\omega^{(k)} \tag{39}$$

This is the Haar measure. We will not have to think about the details of the construction of the Haar measure when implementing the simulation of pure gauge theory on the lattice. However, it is important in more complicated algorithms such as the heat bath algorithm [1], which is not discussed here.

2.2.2 Constructing a gauge invariant lattice action out of the links

We want to construct an action on the link variables $U_\mu(n)$ that live on the lattice Λ with lattice spacing a which approaches the continuum action $S_G^{(\text{cont})} \equiv \exp(i\int d^D x \mathcal{L}_G)$ in the limit $a \to 0$. This discretization can be easily written down by the substitution $\int d^D x \to a^D \sum_{n\in\Lambda}$ with a sum over sites with the right power of the lattice spacing in order to maintain dimensionality:

$$\frac{1}{2g^2}\sum_{\mu,\nu}\int d^4x \text{Tr}\left[F_{\mu\nu}(x)^2\right] \to \frac{a^D}{2g^2}\sum_{\mu,\nu}\sum_{n\in\Lambda}\text{Tr}\left[F_{\mu\nu}(n)^2\right] \tag{40}$$

So our goal is to construct a gauge invariant action $S_G(U)$ on a lattice with lattice spacing a which satisfies $\lim_{a\to 0} S_G(U) = S_G^{(\text{cont})}(A)$. In other words, we need to construct a funcion $S_G(U)$ which satisfies:

$$S_G(U) = \frac{a^4}{2g^2}\sum_{\mu,\nu}\sum_{n\in\Lambda}\text{Tr}[F_{\mu\nu}(n)^2] + \mathcal{O}(a^5) \tag{41}$$

Since we need S_G to be gauge invariant, let's consider the simplest gauge invariant objects which we can construct: the trace of the "placquettes"

$$U_{\mu\nu}(n) \equiv U_\mu(n)U_\nu(n+\hat{\mu})U_{-\mu}(n+\hat{\mu}+\hat{\nu})U_{-\nu}(n+\hat{\nu}) \tag{42}$$

Where

$$U_{-\mu}(n) = U_\mu^{-1}(n-\hat{\mu}) \tag{43}$$

These objects are gauge invariant because under a gauge transformation they transform as:

$$\begin{aligned}
\text{Tr}U_{\mu\nu}^{(\Omega)}(n) &= \text{Tr}[U_\mu^{(\Omega)}(n)U_\nu^{(\Omega)}(n+\hat{\mu})\underbrace{U_{-\mu}^{(\Omega)}(n+\hat{\mu}+\hat{\nu})}_{\left(U_\mu^{(\Omega)}(n+\hat{\nu})\right)^{-1}}\underbrace{U_{-\nu}^{(\Omega)}(n+\hat{\nu})}_{\left(U_\nu^{(\Omega)}(n)\right)^{-1}}] \\
&= \text{Tr}\left[(\Omega(n)U_\mu(n)\Omega(n+\hat{\mu})^{-1})(\Omega(n+\hat{\mu})U_\nu(n+\hat{\mu})\Omega(n+\hat{\mu}+\hat{\nu}))\right. \\
&\quad \times(\Omega(n+\hat{\nu})U_\nu(n+\hat{\nu})\Omega(n+\hat{\nu}+\hat{\mu})^{-1})^{-1}(\Omega(n+\hat{\nu})U_\nu(n+\hat{\nu})\Omega(n+\hat{\nu}+\hat{\mu})^{-1})^{-1} \\
&\quad \left.\times(\Omega(n+\hat{\nu})U_\mu(n+\hat{\nu})\Omega(n+\hat{\nu}+\hat{\mu})^{-1})^{-1}(\Omega(n)U_\nu(n)\Omega(n+\hat{\nu})^{-1})^{-1}\right] \\
&= \text{Tr}U_{\mu\nu}(n)
\end{aligned}$$

Where we have used the facts that $(AB)^{-1} = B^{-1}A^{-1}$ and $\text{Tr}[AB] = \text{Tr}[BA]$. These objects turn out to be all we need to construct an action which satisfies (40). By writing $U_{\mu\nu}(n)$ in terms of the $A_\mu(n)$ fields, one can show that the placquettes are in fact the lattice discretization of the field strength tensor in the continuum:

$$U_{\mu\nu} = \exp(ia^2 F_{\mu\nu}(n) + \mathcal{O}(a^3)) \tag{44}$$

Exercise 1 Verify (44):
i) The Baker-Campbell-Hausdorff formula states that:

$$\exp(A)\exp(B) = \exp\left(A + B + \frac{1}{2}[A,B] + \cdots\right) \tag{45}$$

Where the omitted terms are all of the form $A^n B^m$ with $n + m > 2$. Verify that this expansion holds to this order by expanding both sides in powers of A and B.

ii) Use (45) to expand $U_{\mu\nu}(n)$ in terms of $A_\mu(n)$, neglecting terms of order a^3 or higher.

iii) Taylor expand the A_μ fields like so:

$$A_\mu(n + \hat{\nu}) = A_\mu(n) + a\partial_\nu A_\mu(n) + \mathcal{O}(a^2) \tag{46}$$

and use the continuum form of $F_{\mu\nu}$ to show (44).

Expanding the exponential in (44) gives:

$$\begin{aligned} U_{\mu\nu} &= 1 + \left(ia^2 F_{\mu\nu}(n) + \mathcal{O}(a^3)\right) + \left(ia^2 F_{\mu\nu}(n) + \mathcal{O}(a^3)\right)^2 + \mathcal{O}(a^6) \\ &= 1 + ia^2 F_{\mu\nu}(n) + \mathcal{O}(a^3) - a^4 F_{\mu\nu}^2(n) + \mathcal{O}(a^5) \end{aligned}$$

We can get rid of the coefficient attached to a^2 by taking the real part of $U_{\mu\nu}$, since that coefficient is purely imaginary. It turns out that the coefficient attached to a^3 is also purely imaginary. This means that we can write down a description of $a^4 F_{\mu\nu}(n)$ in terms of the links $U_\mu(n)$ which is good to $\mathcal{O}(a^4)$, which is precisely what we wanted:[4]:

$$\begin{aligned} &\mathrm{Re}\,(U_{\mu\nu}) = 1 - a^4 F_{\mu\nu}^2 + \mathcal{O}(a^5) \\ &\rightarrow a^4 F_{\mu\nu}^2 = 1 - \mathrm{Re}\,(U_{\mu\nu}) + \mathcal{O}(a^5) = \mathrm{Re}\left[1 - (U_{\mu\nu}) + \mathcal{O}(a^5)\right] \\ &\rightarrow a^4 \mathrm{Tr}\left[F_{\mu\nu}^2\right] = \mathrm{ReTr}\left[1 - U_{\mu\nu} + \mathcal{O}(a^5)\right] \end{aligned}$$

So we have:

$$S_G(U) = \frac{2}{g^2}\sum_{n\in\Lambda}\sum_{\mu<\nu}\mathrm{ReTr}[1 - U_{\mu\nu}(n)] \tag{47}$$

[4]My derivation above shows that the lattice artifacts are all of the order a^5 or higher. In [1], Gattringer and Lang write (see equation 2.54) that $S_G(U) = S_G^{(\mathrm{cont})}(U) + \mathcal{O}(a^2)$, which is in disagreement with my derivation. This must be a typo - if terms of order a^2 were lying around, $S_G(U)$ would not approach $S_G^{(\mathrm{cont})}(U)$ in the continuum limit. It's possible that Gattringer and Lang meant that the *relative* error is $\mathcal{O}(a^2)$, i.e. that the a^5 terms are imaginary and/or traceless and therefore do not contribute to S_G, meaning that lattice artifacts die as a^6 or higher, contributing a relative error of a^2 when compared to the a^4 term that stays around in the continuum limit. I don't know if this is the case though, since I haven't computed the a^5 artifacts. It could just be a typo in the book.

Where the sum on the RHS does not include $\nu \geq \mu$ because $U_{\mu\nu} = U^{\dagger}_{\nu\mu}$.

3 Algorithm

Numerical quadrature algorithms to calculate high-dimensional integrals are exponential in the number of dimensions. The dimensionality of integrals like (22) is intractably large due to the fact that there is one SU(3) variable for each link, and there are DV links on the lattice, where D is the dimension of the lattice and V is the volume. For a $D = 4, V = 64^4$ lattice, this results in an integral of dimension $8 \times 4 \times 64^4 = 536,870,912$. In 2018, The US Department of Energy unveiled Summit, which can perform 200 quadrillion calculations per second. To estimate this integral with a numerical quadrature algorithm which scaled like 10^{8DV}, Summit would take about $10^{8DV} \times (1/(200 \times 10^{15} \times 60^2 \times 24 \times 365) \approx 10^{8DV}/10^{25} \approx 10^{5\times 10^9}$ years. Since the Department of Energy isn't going to get exponentially better at becoming exponentially better at building supercomputers anytime soon, we're going to have to resort to something else.

Our goal is to instead come up with a set of instructions which will use a well-defined amount of computational resources to approximate observables of the form (22) as well as an approximation of the error introduced due to the finite amount of resources involved. We also want this approximation to be such that more resources results in a smaller amount of error. This is surprisingly very easy to do.

3.1 Computing observables statistically

Let

$$\rho(U) \equiv \frac{e^{-S_G(U)}}{\int DV e^{-S_G(V)}} \tag{48}$$

Interpreting (48) as probability distribution leads us to such an algorithm immediately, since then (22) is of the form of an expectation value. Naively, an algorithm to approximate $\langle \mathcal{O} \rangle$ and the error involved in doing so is to do the following:

1. Draw N samples $U_i \sim \rho(U)$
2. Approximate (22) by the estimator

$$\langle \mathcal{O} \rangle \approx \langle \mathcal{O}(U_i) \rangle = \frac{1}{N} \sum_{i=1}^{N} \mathcal{O}(U_i) \tag{49}$$

3. We then approximate the variance to give us an approximation of the error involved in (49) :

$$\sigma^2_{\mathcal{O}} \approx \frac{1}{N-1} \sum_{i=1}^{N} (\mathcal{O}(U_i) - \langle \mathcal{O}(U_j) \rangle) \tag{50}$$

Assuming that $\frac{e^{-S_G(U)}\int D\psi\mathcal{D}\bar{\psi}\ e^{-S_F(\psi,\bar{\psi},U)}}{\int DVe^{-S_G(V)}\int D\psi\mathcal{D}\bar{\psi}\ e^{-S_F(\psi,\bar{\psi},V)}}$ can be interpreted as a probability distirbution[5], this procedure can be applied to QCD as well. This algorithm makes clear one of the powers of the path integral formalism, which is to introduce a completely non-perturbative approach to caclulating observables of our field theory. However, we can only do this directly sample from factorizable distributions and a few other special cases. It is unclear how to do the first step of this algorithm at all, because we have no way of drawing a sample U_i from $\rho(U)$, which is a non-factorizable multidimensional distribution which imposes non-trivial correlations between the link variables. One solution to this problem is to use a Markov Chain to generate samples.

3.2 Markov Chain Monte Carlo

Markov chain monte carlo is a statistical method of generating samples which are correlated with some characteristic correlation length τ from a distribution. In a Markov chain, U_k is obtained from U_{k-1} via a stochastic process described by some transition probability function $T(U_{k-1}, U_k)$ that satisfies:

1. $T(U,V) \geq 0\ \forall U,V,\ \int DV\ T(U,V) = 1$
2. $\int DU\rho(U)T(U,V) = \rho(V)$ ("stability")
3. $\forall U,V\ \exists n \mid T^n(U,V) > 0$ ("ergodicity")
4. $T(U,U) > 0\ \forall U$ ("aperiodicity")

The stability condition can be interpreted as saying that for any group element V, if we had a source of independent samples $U \sim \rho(U)$ and allowed them to transition according to $\tilde{\rho}(U) = T(U,V)$, which is a well defined probability distribution due to the first condition, then they would transition to a distribution $\rho(V)$. This is the same as saying that $\rho(V)$ is the *fixed point* of our Markov chain. *Detailed balance* is a sufficient condition for stability:

$$\rho(U)T(U,V) = \rho(V)T(V,U) \tag{51}$$

Which is the statement that if you were to imagine many configurations transitioning at the same time according to $T(U,V)$, the flow of configurations into a site V, which is given by $p(U)T(U,V)$ where $p(U)$ is the current distribution of configurations over the chain, will be balanced by the net flow of configurations leaving the site V if $p(U) = \rho(U)$.

As long as the ergodicity condition is satisfied, the Markov chain will eventually reach this fixed point ρ. The time it takes to do so in practice is generally referred to as the *thermalization time* of the chain.

We can use the samples generated from the markov chian as an approximation of N configurations sampled directly from the distribution, by generating CN correlated configurations with some correlation length $\tau \ll C$ and then discarding every Cth configuration. This leaves us with N samples which look very independent in the sense that

[5]See footnote 3.

if you measured the correlation length of those configurations, it would be exponentially damped by a factor of τ/C.

So our algorihtm to compute $\langle\mathcal{O}\rangle$ with an error which scales like $\sqrt{N}$ is to generate N quasi-independent gauge configurations sampled according to $\exp(-S_G(U))$ by first constructing a transition function $T(U,V)$ which satisfies the properties above and the condition that $p(U_k) = T(U_{k-1}, U_k)$ can be sampled from tractably. Then, we use the transition function to generate samples like so:

1. First, estimate the autocorrelation time τ of the observable, which is defined to be $\tau_{\mathcal{O}}$ such that:[6]

$$\langle\mathcal{O}(U_i)\mathcal{O}(U_{i+t})\rangle \approx \langle\mathcal{O}(U_i)\mathcal{O}(U_i)\rangle \exp\left(-\frac{t}{\tau_{\mathcal{O}}}\right) \tag{52}$$

   By performing the following steps:

   (a) Initialize the Markov chain to an arbitrary gauge configuration U.

   (b) Sample $V \sim T(U,V)$. Store $V \to U$.

   (c) Repeat step 1(b) until $\mathcal{O}(U)$ equillibriates.

   (d) Repeat step 1(b) a few times, saving U at each step to a set $\{U_i\}$.

   (e) Approximate the autocorrelation length $\tau_{\mathcal{O}}$ by computing:

$$\tau_{\mathcal{O}} \approx t \ln \frac{\langle\mathcal{O}(U_i)\mathcal{O}(U_i)\rangle}{\langle\mathcal{O}(U_i)\mathcal{O}(U_{i+t})\rangle} \tag{53}$$

   for a value of t large enough to let you see the autocorrelation stabilize, but not too long that signal-to-noise error destroys the calculation (after some number of configurations the autocorrelation will be practially 0 but there will still be some noise)

   (f) Set $N_{\text{skip}} \gg \tau_{\mathcal{O}}$.

2. Sample $V \sim T(U,V)$. Store $V \to U$. Repeat until $\mathcal{O}(U)$ equillibriates.

3. Repeat the following steps N_{meas} times:

   (a) Sample $V \sim p(V) = T(U,V)$. Store $V \to U$. Do this N_{skip} times.

   (b) Add the current U to the set of gauge configurations $S_{\text{meas}} = \{U_i\}$.[7]

[6] I have not done this step in the code described in these notes - I simply check in a very ad-hoc manner to see if the configurations I get at the end of the metropolis look independent. This is not good practice, and you might want to implement this step yourself before analyzing the metropolis data.

[7] In practice, the metropolis is not actually restarted. The first few configurations in S_{meas} are used as the set S_{autocorr} to compute the autocorrelation, and one can just skip every few configurations if they find that N_{skip} turned out to be too small to remove autocorrelations. The algorithm is just simpler to write down and it is more well-defined if you already know the autocorrelation time, which is why I wrote it in steps like this.

A few points to note:

- This procedure is quite ad-hoc: We have not rigorously defined what it means for the Markov chain to equillibriate. In practice, what it means is that the desired obervable $\mathcal{O}$ starts to fluctuate around a well-defined average value. This does not always happen, such as when taking infinite volume extrapolations around critical points for example, and a more rigorous algorithm and analysis of the data is required in those cases.
- Every step of this procedure is dependent on $\mathcal{O}$, and the transition function T: Some observables will have larger autocorrelation and/or equillibriation times than others etc.

Now we have a way to generate samples, as long as we have a transfer probability function $T(U, V)$ which meets the requirements described above *and* is tractable to sample from. A general scheme for constructing such a function is given by the Metropolis algorithm:

3.3 The Metropolis-Hastings Algorithm

Given a probability distribution $\rho(U)$ which we want to ensure is the fixed point of our Markov chain, we can construct a generic $T(U, V)$ which:

- Satisfies detailed balance (51), and therefore gaurantees stability and the existence of the desired fixed point
- Can be sampled from easily

We can do this by first constructing a transition function $T_P(U, V)$ which simply *proposes* the next configuration in our Markov chain, and does not necessarily satisfy (51). We then accept the configuration with another transition function $T_A(U, V)$ which accounts for the difference between the proposal distribution $T_P(U, V)$ and the true distribution $\rho(V)$ in just the right way so that the total probability of proposing and then accepting the configuration:

$$T^{(\mathrm{MH})}(U, V) = T_P(U, V) T_A(U, V) \tag{54}$$

does satisfy (51). The way to do this is to construct $T_A(U, V)$ like so:

$$T_A(U, V) = \min\left(1, \frac{T_P(U, V)\rho(V)}{T_P(V, U)\rho(U)}\right) \tag{55}$$

Exercise 2
i) Show that the acceptance probability (55) ensures that (54) satisfies detailed balance (51) for any U-dependent proposal distribution $T_P(U, V)$ on the proposals V.

ii) Show that detailed balance implies stability.

Optional: Consider a Markov chain whose states are discrete variables U_n instead of continuous ones $U(\omega)$. Write down the discrete analog of the four conditions described in Section 3.2 and prove that the discrete analog of the stability condition ensures that the discrete analog of $\rho(U)$ is a fixed point of the $n \times n$ discrete transfer matrix which takes the place of the transfer probability function $T(U, V)$. It might help to learn about the Perron-Frobenius theorem.

The Metropolis-Hastings algorithm to calculate this transition probability function is to do the following:

1. Starting with configuration U, sample $V \sim T_P(U, V)$.
2. Generate a random number $r \in [0, 1]$ uniformly.
3. Compare r to $T_A(U, V)$:
   - If $r \leq T_A(U, V)$, accept V: Store $V \to U$.
   - If $r > T_A(U, V)$, reject V: Do nothing.

Recall that the reason we could not use the naive algorithm in Section 3.1 was because we could not sample directly from (48). We can't even *compute* $p(V)$ due to the partition function $\int DV \exp(-S_G(V))$ which itself is a high-dimensional integral. But here lies the beauty of the Metropolis-Hastings algorithm: We don't have to! Because the acceptance probability and therefore the algorithm described above involve computing only ratios of the form $\rho(V)/\rho(U)$, the normalization factor drops out entirely.

Note that if the proposal distribution is *symmetric*, i.e. $T_P(U, V) = T_P(V, U)\ \forall U, V$, then (55) reduces to:

$$T_A(U, V) = \min\left(1, \frac{\rho(V)}{\rho(U)}\right) \tag{56}$$

3.4 Proposing local updates of link variables

Because $S_G(U)$ is local, the Metropolis-Hastings acceptance rate becomes very convenient to calculate if we propose local changes in the gauge configurations during the proposal step of the Metropolis algorithm. We do this by selecting a random matrix $X \in \mathrm{SU}(3)$ and then proposing a gauge configuration V which is altered only locally

from U, at a single link at site n pointing in direction μ:

$$V_\nu(m) = \begin{cases} XU_\nu(m) & \mu = \nu, n = m \\ U_\nu(m) & \text{else} \end{cases} \tag{57}$$

In other words, we are setting the proposal distribution of the Metropolis-Hastings algorithm to be:

$$T_P(U, V) = \sum_{\{X \in S \ | \ V_\mu(n) = XU_\mu(n)\}} \frac{1}{|S|} \tag{58}$$

This is not that hard to compute, but we can get away with not computing it by ensuring that X and X^{-1} have the same probability of being chosen. This makes T_P symmetric and results the Metropolis-Hastings acceptance rate taking the simpler form (56). A scheme of sampling SU(3) matrices which satisfies this requirement is to generate an array of random SU(3) matrices and store their inverses as well, and then uniformly sampling from the set of random matrices and inverse random matrices at each step in the metropolis.

Two hyperparameters, denoted in the code as e and M, are involved in this scheme:

- e is used to decide how close to the identity - and therefore how likely the proposals generated by them are to be accepted - the X matrices should be.[8] Once e is chosen, the method of generating these matrices is described in Section A.3.
- M is the number of matrices generated (which means that the array will store $|S| = 2M$ elements - those matrices and their inverses). I have defined it in the code to scale with the volume, because if M is held fixed, then as the volume of the lattice grows, the probability of selecting the same X for many local proposals will become larger and artificially increase the autocorrelation of the observables.

The locality of $S_G(U)$ comes in handy when we compute the acceptance probability (56) using this proposal distribution, which takes the form:

$$T_A(U, V) = \min\left(1, \frac{\exp(-S_G(V))}{\exp(-S_G(U))}\right) = \min\left(1, \exp(-\triangle S_G)\right) \tag{59}$$

where

$$\triangle S_G \equiv S_G^{(\text{proposed})} - S_G^{(\text{current})} = S_G(V) - S_G(U) \tag{60}$$

The action (47) up to a constant is the sum of the traces of all of the elementary loops. Instead of summing over loops $P_{\mu\nu}$, which makes it hard to understand the effect of a single link changing, we can write it as the sum over all links of the traces of all of the elementary loops which can be formed from that link: Given a link $U_\mu(n)$, there are two ways to form an elementary loop in the $\mu - \nu$ plane, for each direction $\nu \neq \mu$: Start

[8] With higher acceptance rate comes larger autocorrelation, so in practice e is tuned to be around 50%. Changing it doesn't really help very much unless you are looking at phenomena with properties such as the critical slowing down that appears in phase transitions.

following one of the two distinct paths $n \to n+\hat{\mu} \to n+\hat{\mu} \pm \hat{\nu}$ and finish the loop from there, as it is from that point on uniquely identified by tracing your way back to n:

$$n \to n+\hat{\mu} \to n+\hat{\mu} \pm \hat{\nu} \to \begin{cases} n+\hat{\mu} \\ n-\hat{\nu} \end{cases} \to n \tag{61}$$

In other words, S_G can be written as:

$$S_G(U) = \sum_{n,\mu} \mathrm{ReTr}\left[1 - U_\mu(n) \sum_{\nu \neq \mu} S^{(U)}_{\mu\nu}(n)\right] \tag{62}$$

Where

$$S^{(U)}_{\mu\nu}(n) = U_\nu(n+\hat{\mu})U_{-\mu}(n+\hat{\mu}+\hat{\nu})U_{-\nu}(n+\hat{\nu}) + U_{-\nu}(n+\hat{\mu})U_{-\mu}(n+\hat{\mu}-\hat{\nu})U_\nu(n-\hat{\nu}) \tag{63}$$

are the so-called *staples* which finish the construction of the elementary loops in the $\mu - \nu$ plane. With the locality of the action now fully explicit, we can easily compute the action difference induced by the proposal (57):

$$\begin{aligned} \triangle S_G = S(V) - S(U) &= \sum_{m,\sigma} \mathrm{ReTr}\left[1 - V_\sigma(m) \sum_{\nu \neq \sigma} S^{(V)}_{\sigma\nu}(m)\right] - \sum_{m,\sigma} \mathrm{ReTr}\left[1 - U_\sigma(m) \sum_{\nu \neq \sigma} S^{(U)}_{\sigma\nu}(m)\right] \\ &= \sum_{m,\sigma} \mathrm{ReTr}\left[U_\sigma(m) \sum_{\nu \neq \sigma} S^{(U)}_{\sigma\nu}(m) - V_\sigma(m) \sum_{\nu \neq \sigma} S^{(V)}_{\sigma\nu}(m)\right] \end{aligned}$$

Now, because $V_\sigma(m) = U_\sigma(m)$ for all $\sigma, m \neq \mu, n$, and $V_\mu(n) = XU_\mu(n)$, the only term in the sum which survives is:

$$\triangle S_G = \mathrm{ReTr}\left[(U_\mu(n) - XU_\mu(n)) \sum_{\nu \neq \mu} S^{(U)}_{\nu\mu}\right] \tag{64}$$

The reason we decided to make local upates is because of the simple form of $\triangle S_G$ that is gauranteed to occur since S_G is a sum over local contributions. This makes the metropolis algorithm very straightforward to implement, since we can just propose local updates and compute $\triangle S_G$ at each step and then use (59) to gaurantee detailed balance of the Markov chain.

3.5 Putting it all together

Assume that we already know the autocorrelation time $\tau_{\mathcal{O}}$ of the observable $\mathcal{O}$ that we want to measure (which depends on the parameters of the action and possibly on hyperparameters of the markov chain which alter the proposal distribution), and have set $N_{\text{skip}} \gg \tau_{\mathcal{O}}$. Say we also have a good idea of how long it will take for the Markov chain to equillibriate and have set N_{thremo} (as in the number of steps to "thermalize"

when viewed as a statistical ensemble) to be significantly larger than this value. Then the algorithm to obtain an approximation of (22) whose error scales as $N_{\text{meas}}^{-1/2}$, which is the algorithm which I have implemented in the code described in the following section, is as follows:

1. Set U to be an arbitrary gauge configuration.
2. Sample $V \sim T^{(\text{MH})}(U, V)$ by proposing DV local updates and accepting each one with probability $\min(1, \exp(-\triangle S_G(\mu, n)))$.
3. Perform the second step N_{thermo} times.
4. Repeat the following steps N_{meas} times:
   (a) Perform the second step N_{skip} times.
   (b) Store the current U in a set $\{U_i\}$.
5. Output $\frac{1}{N_{\text{meas}}} \sum_{i=1}^{N_{\text{meas}}} \mathcal{O}(U_i)$.

4 Implementation

The first step in our implementation of the algorithm described above will be to think about how to store the link variables in our code and index them for later use. See Section A.1 for a description of how I've done this, which wil be necessary in order to understand how the rest of my implementation works - of course, all methods of doing so are equally valid.

Sample code has been provided with my implementation of the algorithm. Steps 1-4 are implemented in `metropolis.cpp`, which outputs the set $\{U_i\}$ to a file that is then read by `compute_observables.cpp`[9], which handles step 5.

The separation of the code into `metropolis` and `compute_observables`, and the manner in which gauge configurations are sent from the former to the latter, is helpful because it is easy to perform further analysis on the gauge configurations by writing code which calls `read_configuration()` in the same way that `compute_obsevables` does - for example, one could write code to compute the autocorrelation length of the configurations and as long as they do so via calling `read_configuration()`, they can just feed in the data from the metropolis.

Note that here we use Gattringer and Lang's convention for the action:

$$S_G(U) = \frac{\beta}{3} \sum_{n \in \Lambda} \sum_{\mu,\nu} \text{ReTr}\left[1 - U_{\mu\nu}\right] \tag{65}$$

where

$$\beta = \frac{6}{g^2} \tag{66}$$

[9] see Section A.2 for more details about how the configurations are shared between the two processes.

First, we initialize the configuration to be a random gauge configuration of random SU(3) matrices:

```
void initialize(ranlux48& rnd, double e){
       V = 1;
       for(int i = 0; i < D; i++) V *= L[i];
       M = 3*V;
       a = new Matrix3cd[V*D];
       for(int i = 0; i < V*D; i++){
              su3(&a[i], e, rnd);
       }
}
```
su3.cpp

Then, we run the metropolis:

```
int main(int argc, char** argv){
       int iseed, nthermo, nskip, nmeas;
       double e;

       scanf("%d %d %d %lf %lf\n",&nthermo,&nskip,&nmeas,&iseed,&beta,&e);
       scanf("%d ",&D);
       L = new int[D];
       for(int i = 0; i < D; i++)
              scanf("%d",&L[i]);

       printf("nthermo=%d nskip=%d nmeas=%d D=%d L=%d iseed=%d beta=%e e=%e\n",nthermo,
           nskip,nmeas,D,L,iseed,beta,e);

       ranlux48 rnd(iseed);
       initialize(rnd, e);
       genX(rnd, e);
       idist = uniform_int_distribution<int>(0,2*M-1);

       for(int i = 0; i < nthermo; i++){
              update(rnd);
              genX(rnd,e);
       }

       for(int i = 0; i < nmeas; i++){
              for(int j = 0; j < nskip; j++) update(rnd);
              genX(rnd,e);
              printf("U: "); print();
       }
}
```
metropolis.cpp

This code relies on a method `update()` which performs Step 2 of the algorithm described in Section 3.5. In the code provided, this method has not been explicitly implemented in `metropolis.cpp`, but rather implemented in another file and precompiled into the object file `update.o`. This is because one of the exercises is to implement it yourself (see below). If you want to edit other parts of the code and compile it without implementing this, you will need to link a precompiled file which contains an implementation

of this method. You can do this by running `make -f make_with_update_implemented` instead of `make` when compiling the code.

Finally, we read the configurations from the metropolis into `compute_observables.cpp` and then compute observables such as the placquette energy and the average polyakov loop (see Section 6:

```
void plaquette(Matrix3cd *g, int i, int d1, int d2)
{
	*g = Matrix3cd::Identity();
	*g = *g*a[i*D+d1];
	*g = *g*a[step(i,d1,1)*D+d2];
	*g = *g*(a[step(i,d2,1)*D+d1].inverse());
	*g = *g*(a[i*D+d2].inverse());
}
```
su3.cpp

```
void polyakov(Matrix3cd *g, int i)
{
	*g = Matrix3cd::Identity();
	int i0 = i;
	do {
		*g = *g*a[i*D+0];
		i = step(i,0,1);
	} while (i0 != i);
}
```
compute_observables.cpp

Bootstrap error estimation is then performed on all of the observables like so:

```
void bootstrap(double *mean, double *stdev, double *dat, int len)
{
	*mean = 0;
	for (int i = 0; i < len; i++)
		*mean += dat[i]/len;
	double var = 0.;
	for (int b = 0; b < B; b++) {
		double meanp = 0.;
		for (int i = 0; i < len; i++)
			meanp += dat[rand()%len]/len;
		var += (meanp-*mean)*(meanp-*mean)/B;
	}
	*stdev = sqrt(var);
}
```
compute_observables.cpp

5 Exercises with the code

Exercise 3 Edit, compile, and run the code:
Compute the average determinant of each configuration U, and print it out as a function of metropolis step time[a] at each step of the metropolis. Check to see if floating point errors move the configurations away from the SU(3) manifold, which would result in a non-unity value of the average determinant of the matrices in the configurations being sampled from the metropolis, and also check that `compute_observables` still reads in gauge configurations correctly and gives you the same results despite this extra information being printed out into the file which you use to store the output of the metropolis. Do this for a 4^4 lattice with $\beta = 2.0$ and $e = 0.5$. Choose $N_{\text{skip}} = 1$, $N_{\text{therm}} = 0$, and $N_{\text{meas}} = 1000$ to see if this error shows up by the 1000th update, for example.

[a] This can be done by iterating through every link in the configuration (as is done in `print()` and adding the determinant of the matrix living at that link to a complex variable `average_determinant`, and then running `printf("AVERAGE DET: %e\n",average_determinant /(D*V));`, because there are DV links in a lattice of dimension D with volume V.

Exercise 4 Implement the portion of the `metropolis.cpp` code which ensures that the fixed point of the Markov chain will be the desired probability distribution $\rho(U) = \exp(-S_G(U))/\int DV \exp(-S_G(V))$ over the link variables yourself:

1. Implement `int update(ranlux48& rnd)` at the start of `metropolis.cpp`:
   (a) Iterate over all of the links. You can do this by iterating over all of the sites indexed by $i \in \{0, 1, \cdots, V-1\}$, where $V = |\Lambda| = \prod_{i=1}^{D} L_i$ (which is computed at the start of the metropolis code and is a global variable so can be accessed within your method), and then iterating over all of the directions indexed by $\mu \in \{0, 1, \cdots, D-1\}$.[a]
   (b) For each link, sample a random matrix X by simply picking a matrix uniformly from the array `x` of size $2M$, using the `idist` distribution which is defined to be a uniformly distributed distribution of integers on $[0, 2M-1]$.
   (c) Compute the local change in action $\triangle S$ induced by the change $U_\mu(n) \to XU_\mu(n)$ (64). This is will require you to use the `step()` function which is described in Section A.1 in order to compute the stapes (63) associated with the link $U_\mu(n)$.
   (d) Sample a random number r from the interval $[0, 1]$ using the distribution `rdist`, and perform the acceptance step $U_\mu(n) \to XU_\mu(n)$ if $r \leq \triangle S$ and iterate a counter variable. Otherwise, do nothing.
   (e) Return the counter variable, i.e. the number of local proposals accepted during the "sweep"

2. Run `make` to compile the code and see if it works by running

```
./metropolis < input_metropolis > metropolis_data
```

3. Run

```
./compute_observables < metropolis_data > results
```

   for different values of β on a 12^4 lattice, and comparing the results with Gattringer and Lang's plot [1] - see Figure 4.2 and the file `gattringer_lang_average_placquette_energy` provided in the `code` folder which contains the raw data in the plot.

4. If it does not agree, debug! Here are some checks to give you a clue into what might have gone wrong:

   - Look at the acceptance rate and see if it thermalizes to a reasonable value.
   - Check to see if the average placquette energy is thermalizing.
   - Plot the average $\triangle S$ value proposed and see if it varies with the acceptance rate as you expect.
   - Do the following exercises with your own implementation instead of the code provided, and see at what point you get different results

[a] You can see how this iteration is done in the `print()` method, for example, which is implemented in the same file.

Exercise 5 Markov Chain Monte Carlo techniques:
i) Compute the running acceptance rate $\mathcal{A}$ as a function of metropolis time τ, which we define to mean:

$$\begin{aligned}\mathcal{A}(\tau) &\equiv \frac{\text{local proposals accepted during update } U(\tau) \to U(\tau+1)}{\text{local proposals made during update } U(\tau) \to U(\tau+1)} \\ &= \frac{\text{local proposals accepted during update } U(\tau) \to U(\tau+1)}{N_{skip} DV}\end{aligned}$$

Plot $\mathcal{A}(\tau)$ for $\beta = 1.0$ with $N_{\text{skip}} = 10$ and $e = 0.5$ on a 4^4 lattice. Let $N_{\text{therm}} = 0$ so you can see the thermalization yourself, and let N_{meas} be however long it takes to see it. How do you expect $\mathcal{A}(\tau)$ to change as N_{skip} is varied? Is that what happens if you do it? What happens as β is varied?

ii) Based on the definition of the acceptance rate in the Metropolis algorithm (59), how should it be related to the average proposed change in action

$\langle \triangle S \rangle$? Edit the code to print out $\triangle S$ and compute $\mathcal{A}$ and $\langle \triangle S \rangle$ for a few values of e, and then plot $\mathcal{A}(\langle \triangle S \rangle)$. Does the relationship look like what you would expect? What does this mean about the limitations of Metropolis-Hastings? Think about what the average change in action intuitively tells us about the autocorrelation time of observables.

iii) Plot the average placquette energy as a function of metropolis time as well, on the same lattice. Note that `compute_observables.cpp` can be easily modified to print out the value of the placquette energy as it computes it for each τ.

iv) Guess at the "thermalization" times of $\mathcal{A}(\tau)$ and $E(\tau)$ by eye from looking at the plots. Is the acceptance rate a good "observable" to check whether the Markov chain is "thermalized"?

A tip for debugging: Since the configurations being printed and therefore the outputs produced by running the metropolis code are very large, it is necessary to have a way to print information useful for debugging (such as the local change in the action, the current acceptance rate, the matrix elements of the staple, etc.) and access the output without combing through the lines being produced by the metropolis. This is easily done by printing out a flag before the information you need to access, like `printf(''DEBUG INFO: %d\n'',debugging_information)`, for example, and then running `./metropolis < input_metropolis > output_file; grep ''DEBUG INFO: '' output_file`. This is especially helpful when you need to plot the data, because you can do `grep ''DEBUG INFO: '' output_file > debug_file;` and then plot the contents of `debug_file` using your favorite method without having to tell your plotting script which lines you want to plot. I recommend doing precisely this in order to plot the action as a function of metropolis time as an initial check of thermalization. This extra output will not affect computation of observables as long as `''DEBUG INFO''` is not equal to `''U''` - see Section A.2 for an explanation of why this is.

6 Learning about confinement from pure gauge theory

Out of all the possible models of strong interactions, the reason we ended up studying the SU(3) gauge theory N fermionic fields in the fundamental representation coupled to $N^2 - 1$ scalar fields transforming in the adjoint representation of $N = 3$ is due to the UV and IR physics that it predicts:

1. UV: We saw earlier after tedious computation of the running of the coupling in the perturbative regime that if $N = 3$, then as long as $n_f < 16$ the virtual quark-antiquark pairs and the vacuum polarization of the gluons arising from loop corrections will result in the beta function being negative and therefore a well defined,

UV-complete theory of interactions at high energies.

2. IR: The same effect causes the coupling to increase and eventually leave the perturbative regime at lower energy scales. This is actually a good thing, because it is a signal that the physics of this theory is non-perturbative at lower energy scales and in fact it does exhibit non-perturbative properties like chiral symmetry breaking and *confinement*: The fact that all stable bound states in nature are color singlets, despite the triplet nature of the quarks which the theory is built from.

While pure gauge theory can't directly tell us about the nature of confinement in the full theory of QCD, since we do not have dynamical quarks, we can still make claims about infinitely heavy, or "static", quarks. The confinement of static quarks is a singificantly distinct phenomena than that of finitely heavy quarks due to the fact that quark-antiquark pairs can only be produced in the latter case, but it is nevertheless a window into the important and mysterious properties of confinement.

The simulation code described here can be easily extended to explore various features of pure gauge theory that can tell us about confinement in a variety of different ways. Some ideas for what to do are:

6.1 Computation of the static potential

Consider a lattice with spacing a and periodic time extent N_T. The *Polyakov loop* is a Wilson loop with non-zero winding number in the time direction:

$$P(\boldsymbol{n}) = \mathrm{Tr}\left[\prod_{j=0}^{N_T-1} U_4(\boldsymbol{n}, j)\right] \tag{67}$$

Where $U_4(\boldsymbol{n}, j)$ denotes the timelike link at the lattice site at spatial point $\boldsymbol{n}$ at time j. Polyakov loops are gauge invariant due to periodic boundary conditions resulting in the loop representing a closed path in periodic time which runs through the spatial site $\boldsymbol{n}$. The potential energy of two infintiely massive quarks separated by distance r is related to the expectation value of the correlation function of two polyakov loops at points $\boldsymbol{n}$ and $\boldsymbol{m}$ such that $r = a|\boldsymbol{m} - \boldsymbol{n}|$ like so [1]:

$$\langle P(\boldsymbol{m}) P(\boldsymbol{n})^\dagger \rangle \propto e^{-N_T a V(r)}(1 + \mathcal{O}(e^{-N_T a \triangle E})) \tag{68}$$

Where $\triangle E$ is the difference in potential energy between the state of two infinitely heavy quarks and that of the next highest energy state with the same quantum numbers. Computing the polyakov loop correlator will allow us to extract the static quark potential energy $V(r)$, which we expect to be parametrized by

$$V(r) = A + \frac{B}{r} + \sigma r \tag{69}$$

Because at weak coupling (large β), the gauge fields are abelian and we expect to see the $\frac{1}{r}$ potential of QED:

$$\lim_{g\to 0} F_{\mu\nu}^{(\mathrm{QCD})} = \lim_{g\to 0} (\partial_\mu A_\nu - \partial_\nu A_\mu - g[A_\mu, A_\nu]) = \partial_\mu A_\nu - \partial_\nu A_\mu = F_{\mu\nu}^{(\mathrm{QED})} \tag{70}$$

While at strong coupling (small β), we expect to see a linearly rising term that we can focus on in order to extract the *string tension* $\sigma \equiv 900$ MeV/fm. The string tension can be extracted via a computation which builds off of the polyakov loops that have been implemented in Section 4 and can then be compared with the predictions from a small β expansion [1]:

$$\sigma = -\frac{1}{a^2} \ln\left(\frac{\beta}{18}\right) + \mathcal{O}(\beta \ln \beta) \tag{71}$$

6.2 Direct visualization of the flux tube which confines static quarks

The action density

$$s(\boldsymbol{n}) = \frac{1}{2}\mathrm{Tr}\left[F_{\mu\nu}(\boldsymbol{n})^2\right] \tag{72}$$

The topological charge density

$$q(\boldsymbol{n}) = \frac{g^2}{32\pi^2}\epsilon_{\mu\nu\rho\sigma}\mathrm{Tr}\left[F_{\mu\nu}(\boldsymbol{n})F_{\rho\sigma}(\boldsymbol{n})\right] \tag{73}$$

is a measure of the winding of the gluon field lines. When these quantities are plotted on a 3-dimensional spatial slice of the 4D lattice by rendering areas of intense action density in red and areas of moderate action density in blue, for example, they can show us the structure of typical vacuum gluon-field configurations. One can also introduce a static quark source pair into the action easily, since a single Polyakov loop is the world-line of a static quark propogating only in the time direction, as $P(\boldsymbol{x})$ is simply the introdution of a static quark current $j_\mu^{(\boldsymbol{x})}(\boldsymbol{z}) = (0,0,0,1)\delta(\boldsymbol{z}-\boldsymbol{x})$ to the field A_μ:

$$P(\boldsymbol{x}) = \mathrm{Tr}\left[\mathrm{P}\exp\left(i\int d^4z j_\mu^{(\boldsymbol{x})}(z)A^\mu(z)\right)\right] \tag{74}$$

And therefore be able to look at the structure of gluon-field configurations in the presence of such a pair. One can then measure the width of the flux tube [5] and see if it is indeed linearly increasing - which will of course give the same results as the computation of the static potential. However, using $s(\boldsymbol{n})$ and $q(\boldsymbol{n})$, you can now visualize the flux tube and see the actual gluon configurations with your own eyes!

6.3 Observation of the deconfinement phase transition

A non-zero expectation value for $P(\boldsymbol{n})$ intuitively tells us that the cost in free energy of adding a static quark is finite. In a confining phase, this free energy cost is infinite and $\langle P(\boldsymbol{n})\rangle = 0$. At finite temperature, a phase transition occurs which gives $P(\boldsymbol{x})$ a non-vanishing expectation value in pure gauge theory. This corresponds to the breaking of a $\mathbb{Z}_N$ order parameter [2] which can be observed by varying L_T on an $L_S^3 \times L_T$ lattice and measuring the expectation value of $P(\boldsymbol{n})$ as $L_T < L_S$ is varied.

There are various other observables which can be measured that give different insights into confinement as well [2].

A Implementation details

A.1 An elegant scheme of indexing links in an anisotropic lattice

There's a way to only think about the indexing scheme only once and then forget about it when writing lattice code,[10] which is to come up with a one-to-one indexing function

$$i(\vec{n}) : \Lambda \to \mathbb{Z}^{\geq 0} \tag{75}$$

that labels the tuple $\vec{n} = (n_0, n_1, \cdots, n_{D-1})$ associated with a point in a D-dimensional hyper-rectangular lattice Λ with dimensions $\{L_i\}_{i=0}^{D-1}$. If this is cleverly done so that one can easily write a function

$$f(i, d, s) : \mathbb{Z}^{\geq 0} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}^{\geq 0} \tag{76}$$

which returns the index associated with the lattice point that is a signed distance of s away in direction $d \in \{0, 1, \cdots, D-1\}$ from the point indexed by i.[11] Once this function is written, you no longer have to think about the specific choice of indexing scheme $i(\vec{n})$ when computing local quantities such as the difference in the gauge action produced by a local update, for example, because those can be written in terms of this step function.

A convenient choice for $i(\vec{n})$ is one which simply counts upwards from $i(\vec{0}) = 0$ along each of the directions in order of their labels:[12]

$$i(\vec{n}) = \sum_{i=0}^{D-1} n_i \prod_{j=0}^{i-1} L_j \tag{77}$$

Now, a step in direction d with signed distance s is encoded in the shift $n_d \to (n_d + s + |s|L_d) \bmod L_d$, assuming periodic boundary conditions.[13] This means that we need a step function which satisfies

$$f(i(\vec{n}), d, s) = i(\vec{n} + s\hat{e}_d) = \sum_{i \neq d}^{D-1} n_i \prod_{j=0}^{i-1} L_j + ((n_d + s + |s|L_d) \bmod L_d) \prod_{j=0}^{d-1} L_j \tag{78}$$

This can be accomplished with the following function:

[10] All credit here goes to Scott Lawrence, who came up with this scheme

[11] In our case, we have D links pointing in each direction at every site, so we will actually be using this indexing scheme to address the link at site $\vec{n}$ pointing in direction μ by associating it with the index $Di(\vec{n}) + \mu$.

[12] It is interesting to note that in the isotropic case, this indexing function becomes the base 10 representation of the base L number with D digits given by $\vec{n}$.

[13] the last term in the sum ensures positivity of the argument of the remainder operation in the case that $s < -n_d$.

$$f(i,d,s) = \underbrace{\left\lfloor \frac{i}{\prod_{i=0}^{d} L_i} \right\rfloor \prod_{i=0}^{d} L_i}_{\sum_{i>d} n_i \prod_{j=0}^{i-1} L_j} + \underbrace{\left(i + (s + |s|L_d) \prod_{i=0}^{d-1} L_i \right) \bmod \left(\prod_{i=0}^{d} L_i \right)}_{\sum_{i<d} n_i \prod_{j=0}^{i-1} L_j + ((n_d + s + |s|L_d) \bmod L_d) \prod_{j=0}^{d-1} L_j} \tag{79}$$

And this is exactly what I've done in my code:

```
int step(int i, int d, int s) {
	int under = 1;
	for (int i = 0; i < d; i++) {
		under *= L[i];
	}
	return (under*L[d])*(i/(under*L[d])) + (i+under*s+abs(s)*under*
	    L[d])%(under*L[d]);
}
```
su3.cpp

Calculating non-local observables might require knowledge about the indexing scheme, but most can actually be written in terms of only the step function as well.[14] For an example of a non-local observable computed using the step function, see the polyakov loop computation described in Section 4, which can be found in `compute_observables.cpp`.

A.2 Printing out and reading in gauge configurations

The configurations are printed out to the standard output after each metropolis step[15] by calling the method `print()` after each step:

```
void print()
{
	for(int i = 0; i < V; i++){
		for(int j = 0; j < D; j++){
			for(int k = 0; k < 3; k++){
				for(int l = 0; l < 3; l++){
					printf("%e %e ",U[i*D+j](k,l).real(),a[i*D+j](k,l).
					    imag());
				}
			}
		}
	}
	printf("\n");
}
```
metropolis.cpp

The `main()` method of the metropolis code prints out the current gauge configuration (and flushes the standard output buffer to ensure that they are printed before continuing) after every N_{skip} updates, along with a tag "U:" which precedes it to signify that it

[14] For example, correlators are just many steps in the timelike direction!

[15] This output is generally then redirected by the user to the desired output file via running the command `./metropolis < input_metropolis > desired_output_file`. See Section A.4.

is a gauge configuration:

```
for(int i = 0; i < nmeas; i++){
    for(int j = 0; j < nskip; j++) update(rnd);
    printf("U: "); print();
}
```
metropolis.cpp:main()

The configurations are read in by `compute_observables` via `read_configuration()`, which runs through the output of `metropolis`[16], stopping when it sees a line beginning with "U:" and reading it into an array `a` which stores the current configuration being processed:[17]

```
bool read_configuration() {
       char c, c1, c2, c3;

       // skip all lines until gauge configuration is found

       while(true){
              if(feof(stdin)) return 0;
              if((fscanf(stdin, "%c%c%c", &c1, &c2, &c3) == 3) && c1 == 'U' && c2 == ':'
                   && c3 == ' ') {
                    break;
              } else {
                    do{
                          c = fgetc(stdin);
                    }
                    while (c != '\n' && !feof(stdin));
              }
       }

       // read gauge configuration

       for (int i = 0; i < V*D; i++)
              for(int j = 0; j < 3; j++)
                    for(int k = 0; k < 3; k++){
                          if (fscanf(stdin, "%lf %lf ", &re_val, &im_val) != 2)
                              return 0;
                          U[i](j,k) = cd(re_val, im_val);
                    }
       return 1;
}
```
compute_observables.cpp

`read_configuration()` will output 0 once it reaches the end of the output, which is treated as a false value in C++ , so `compute_observables` simply keeps calling it and computing observables on the configuration stored in the array `U` until `read_configurations()` returns 0:

[16] Which is fed to it via a command similar to `./compute_observables < desired_output_file > results`. See section A.4

[17] All credit here goes to Scott Lawrence, who came up with this way of reading configurations easily.

```
while(read_configuration()){
    /* compute functions of the gauge configuration stored in 'U' */
}
```
compute_observables.cpp:main()

Since `fflush(stdout)` is being called in the metropolis code after each call to `print()`, the metropolis code will print out gauge configurations as it samples them.[18] Since the code is split up into separate processes to run the metropolis and compute observables from it, observables can be computed while the metropolis is running instead of only after it has finshed. One can then look at the error on the results and perform an analysis of the autocorrelation length of the configurations etc. while the metropolis is still running, which is very valuable information as it might signify that hyperparameters need to be adjusted or the code needs to be debugged etc. without waiting for the entire run to finish.

One can also edit the metropolis code if desired in order to print out statistics, acceptance rates, debugging information, etc. at each step of the metropolis, without messing up the way the configuration data is going to be read by `compute_observables`, as long as "U:" is not printed before the extra output.

A.3 Generation of random SU(3) matrices

The goal is to create a set of SU(3) matrices which are "close" to the identity in some sense that is characterized by a paramter e, which we can then use to determine the acceptance rate of our Metropolis algorithm. We will do this by constructing them out of SU(2) matrices dependent on e.

To do this, we first write a method to generate random SU(2) matrices. This is done by sampling four random numbers $r_i \in (-\frac{1}{2}, \frac{1}{2}$, and then using them as parameters for the SU(2) matrix

$$\theta(r_0)\sqrt{1-e^2}\mathbb{1} + i\frac{e}{|r|}r \cdot \sigma \tag{80}$$

where e is given as an input into the code, $r = (r_1, r_2, r_3)$, and $\sigma = (\sigma_x, \sigma_y, \sigma_z)$ denotes the pauli matrices:

```
void su2(Matrix2cd *ret, double e, double r0, double r1, double r2, double r3)
{
        double mag = sqrt(r1*r1 + r2*r2 + r3*r3);
        *ret = Matrix2cd::Identity();
        *ret *= abs(r0)/r0*sqrt(1-e*e);
        *ret += e/mag*cd(0,1)*(r1*s_x + r2*s_y + r3*s_z);
}
```
su3.cpp

This method is called three times with a pseudo-random number generator, resulting in three SU(2) matrices which we will denote as r, s, and t. The $\epsilon-$dependent SU(3) matrix $X(e)$ is then constructed like so:

[18] If `fflush(stdout)` is no called, the code does not gaurantee that anything will be printed out until it has finished running the entire metropolis - which might take a long time.

$$X = \begin{pmatrix} r_{11} & r_{12} & 0 \\ r_{21} & r_{22} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} s_{11} & 0 & s_{12} \\ 0 & 1 & 0 \\ s_{21} & 0 & s_{22} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & t_{11} & t_{12} \\ 0 & t_{21} & t_{22} \end{pmatrix} \tag{81}$$

All of this is done in the method `su3()`, which takes a pseudo-random number generator, a value for e, and stores the resulting SU(3) matrix into the parameter g which points to memory allocated for the 3x3 matrix

```
void su3(Matrix3cd *g, double e, ranlux48& rnd){
        uniform_real_distribution<> rdist(-0.5,0.5);
        Matrix2cd r, s, t;
        Matrix3cd R, S, T;

        su2(&r, e, rdist(rnd), rdist(rnd), rdist(rnd), rdist(rnd));
        su2(&s, e, rdist(rnd), rdist(rnd), rdist(rnd), rdist(rnd));
        su2(&t, e, rdist(rnd), rdist(rnd), rdist(rnd), rdist(rnd));

        R.block<2,2>(0,0) = r; R(2,2) = 1;
        S(0,0) = s(0,0); S(0,2) = s(0,1); S(1,1) = cd(1,0); S(2,0) = s(1,0); S
            (2,2) = s(1,1);
        T.block<2,2>(1,1) = t; T(0,0) = 1;

        *g = R*S*T;
}
```
su3.cpp

In my code, I've created an array of matrices `x` which are generated at the start of the metropolis and after each update (see Section 3.4), which contain M randomly generated matrices as well as all of their complex conjugates, which is required in order for the transition probability described in notes above to satisfy detailed balance:

```
void genX(ranlux48& rnd, double e)
{
        x = new Matrix3cd[2*M];
        for(int i = 0; i < M; i++){
                su3(&x[i], e, rnd);
                x[i+M] = x[i].conjugate().eval().transpose();
        }
}
```
su3.cpp

A random matrix is then chosen from `x` when making a local proposal during the metropolis steps. The group elements which are produced by this scheme are not uniformly distributed on the manifold of $SU(3)$ elements in any notion of the word. Such uniformity is not required, because the accept-reject step of the Metropolis algorithm is what will ensure detailed balance when it comes time to simulate link variables according to $S_G(U)$. Intuitively, what's happening is that the "bad" proposals due to $SU(3)$ elements that push us in directions not favored by $\exp(-S_G(U))$ are always going to be

rejected with just the right probability to ensure that $\exp(-S_G(U))$ is the steady-state distribution of the Markov chain[19].

To generate random numbers in production-level code in which you might want to run several copies of at the same time, one might want to seed the pseudo-random number generator automatically using the time and process ID of the job in order to ensure a different source of "random numbers is being used for each job, via something like `srand(time(NULL)+getpid());` being placed at the start of the code instead of taking an input initial seed as my code described here does.

A.4 How to edit and run the code (and C++ Resources)

To compile the code, do the following:[20]

- If you haven't implemented the `update()` method, run: `make -f makefile_with_update_implemented` to compile the code. If you have implemented it, run `make`. To see how this works, see an introduction to makefiles.
- To run the code, run

  ```
  ./metropolis < input_metropolis > metropolis_data
  ```

  and then

  ```
  ./compute_observables < metropolis_data > results
  ```

  To gain more insight into how the file redirection works, see this introduction to input/output redirection.

The following resources contain lots of helpful information about C++:

- Introduction to C++
- Introduction to pointers / How arrays are stored in C++
- Introduction to matrix manipulations in C++ with the Eigen library

[19] Some questions were asked in the lecture about how we can be sure that this scheme produces ergodic proposals, which is necessary for this argument to hold, and even though my hunch is that ergodicity is gauranteed for $e > 0$ and $M > 1$, I'm not sure how to go about proving this. One would need to show that given two SU(3) elements X, Y which are not equal to the identity, there exists an n and m such that any element on the manifold can be expressed as $X^n Y^m$

[20] These directions apply to linux, Macintosh, or the Linux Bash Shell on Windows 10.

References

[1] Christof Gattringer and Christian B Lang. *Quantum Chromodynamics on the Lattice*. Springer, 2009.

[2] Jeff Greensite. The confinement problem in lattice gauge theory. *Progress in Particle and Nuclear Physics*, 51(1):1–83, 2003.

[3] Arthur Jaffe and Edward Witten. Quantum yang-mills theory. *The millennium prize problems*, (1):129, 2006.

[4] Shin Muroya, Atsushi Nakamura, Chiho Nonaka, and Tetsuya Takaishi. Lattice qcd at finite density: an introductory review. *Progress of theoretical physics*, 110(4):615–668, 2003.

[5] Georges Ripka. *Dual superconductor models of color confinement*, volume 639. Springer Science & Business Media, 2004.