Markov Chain Monte Carlo for Dummies

Masanori HANADA

hanada@yukawa.kyoto-u.ac.jp

abstract

This is an introductory article about Markov Chain Monte Carlo (MCMC) simulation for pedestrians. Actual simulation codes are provided, and necessary practical details, which are skipped in most textbooks, are shown. The second half is written for hep-th and hep-lat audience. It explains specific methods needed for simulations with dynamical fermions, especially supersymmetric Yang-Mills. The examples include QCD and matrix integral, in addition to SYM.

Contents

1 Introduction

Markov Chain Monte Carlo (MCMC) simulation is a very powerful tool for studying the dynamics of quantum field theory (QFT). But in hep-th community people tend to think it is a very complicated thing which is beyond their imagination [1]. They tend to think that a simulation code requires a very complicated and long computer program, they need to hire special postdocs with mysterious skill sets, very expensive supercomputers which they will never have access are needed, etc. It is a pity, because MCMC is actually (at least conceptually) very simple,[1] and a lot of nontrivial simulations can be done by using a laptop. Indeed I have several papers for which the coding took at most an hour and crucial parts of simulations were done on a laptop, e.g. [2, 3, 4]. You can quickly write a simple code, say a simple integral with the Metropolis algorithm, and it teaches you all important concepts.

Of course for certain theories we have to invest a lot of computational resources. If you wanted to compete with lattice QCD experts, a lot of sophisticated optimizations,

---

[1]Because I don't like black boxes, I usually code everything by myself from scratch. Still it is extremely rare to use anything more than $+, -, \times, \div, \sin, \cos, \exp, \log, \sqrt{\ }$, "if" and loop. Sometimes a few linear algebra routines from LAPACK [5] are needed, but you can copy and paste them. For the Matrix Model of M-theory [6, 7] you don't even need LAPACK. In short: *nothing more than high school math is needed*. We just have to remove bugs patiently.

sometimes at the level of hardware, would be needed. However there are many other subjects — including many problems in hep-th field — which are not yet at that stage.

There are many sophisticated techniques which enables us to perform large scale simulations with realistic computational resources. They are sometimes technically very complicated but almost always conceptually very simple. It is not easy to invent new techniques by ourselves, but it is not hard to learn and use something experts have already invented. In case you have to do a serious simulation, you may not be able to code everything by yourself. But you can use open-source simulation codes,[2] or you can work with somebody who can write a code. And running the code and getting results are rather straightforward, once you understand the very basics.

In this introductory article, I will present basic knowledge needed for the Monte Carlo study of SYM. I have two kinds of audience in mind: string theorists who have no idea what is lattice Monte Carlo, and lattice QCD practitioners who know QCD but not SYM. For the former, I provide plenty of examples, including sample codes, which are sufficient for running actual simulation codes for SYM. (Sec. 2, Sec. 3 and a part of Sec.4 would be useful for much broader audience including non-physicists.) These materials can also be useful for students and postdocs already working with MCMC; the materials presented here are something all senior people working in MCMC expect their students/postdocs to know, but many students/postdocs do not have chance to learn. I also explain the technical differences between lattice QCD and SYM simulations, which are useful for both string and lattice people.

Note

This version is (probably) not final; more examples and sample codes will be added. I have decided to post it to arXiv because lately I am too busy and do not have much time to work on this.

Sample codes can be downloaded from GitHub, `https://github.com/MCSMC/MCMC_sample_codes`. The latest version of this review will be uploaded there as well.

Comments, requests and bug/typo reports will be appreciated.

2   Markov Chain Monte Carlo (MCMC)

Suppose we want to perform a Euclidean path-integral with a partition function

$$Z = \int [d\phi] e^{-S[\phi]}, \tag{1}$$

---

[2]See e.g. [8, 9, 10, 11] for supersymmetric theories.

where the action $S[\phi]$ depends only on bosonic field(s) ϕ. (In later sections I will explain how to include fermions.) Usually we are interested in the expectation values of operators,

$$\langle \mathcal{O} \rangle = \frac{1}{Z} \int [d\phi] e^{-S[\phi]} \mathcal{O}(\phi). \tag{2}$$

In order to make sense of this expression, typically we regularize the theory on a lattice, so that the path-integral reduces to a usual integral with respect to finitely many variables. Let's call these variables $x_1, x_2, \cdots, x_p$. Typically, the lattice action is so complicated that it is impossible to estimate the integral analytically. Because we have to send p to infinity in order to take the continuum limit or the large-volume limit, a naive numerical integral does not work either. If we approximate the integral by a sum by dividing the integral region of each x_i to n intervals like in Fig. 1, the calculation cost is proportional to n^p, simply because we have to take a sum of n^p numbers. This is hopelessly hard for realistic values of n and p. Suppose $n = 100$ and $p = 10$. Then we have to take a sum of 10^{20} numbers. Let us convince my collaborators in Livermore Laboratory that this integral is extremely important, and use their supercomputer *Sequoia*, which was the fastest in the world from 2012 to 2013. Its performance is 20PFLOPS, namely it can process 2×10^{16} double-precision floating-point arithmetics every second. Let's ignore the cost for calculating the value of the function at each point, and consider only a sum of given numbers. (This is an unrealistic assumption, of course.) But already it takes $10^{20}/(2 \cdot 10^{16}) = 5000$ seconds. Well, it may be acceptable... but if you take $p = 15$, it takes $10^{30}/(2 \cdot 10^{16}) = 5 \times 10^{13}$ seconds, which is about 634,000 years. For a 4d pure SU(3) Yang-Mills on lattice with 10^4 points, p is $4 \times (3^2 - 1) \times 10^4$. We cannot even take $n = 2$. This is the notorious *curse of dimensionality*; when the dimension p is large it is practically impossible to scan the phase space.

Perhaps when you are reading this article you have access to much better machines, but it will not give you much gain.
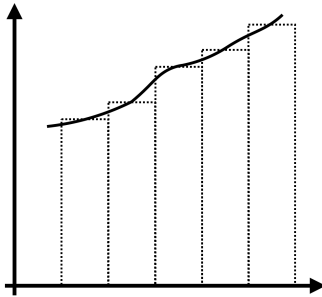

Figure 1: Approximate the integral by a sum of the area of rectangles.

Markov Chain Monte Carlo (MCMC) circumvents the curse of dimensionality based on the idea of *importance sampling*. In most cases of our interest, the majority of the phase space is irrelevant because the action S is large and the weight e^{-S} is very small. If we can

find *important* regions in the phase space and invest our resources there, we can avoid the curse of dimensionality. MCMC enables us to actually do it.

We assume $S[x_1, x_2, \cdots, x_p]$ is real and the partition function $Z = \int dx_1 \cdots dx_p e^{-S[x_1, x_2, \cdots, x_p]}$ is finite. In MCMC simulations, we construct a chain of sets of variables $\{x^{(0)}\} \to \{x^{(1)}\} \to \{x^{(2)}\} \to \cdots \{x^{(k)}\} \to \{x^{(k+1)}\} \to \cdots$ satisfying the following conditions:

- **Markov Chain**. The probability of obtaining $\{x^{(k+1)}\}$ from $\{x^{(k)}\}$ does not depend on the previous configurations $\{x^{(0)}\}, \{x^{(1)}\}, \cdots, \{x^{(k-1)}\}$. We denote this transition probability by $T[\{x^{(k)}\} \to \{x^{(k+1)}\}]$.

- **Irreducibility**. Any two configurations are connected by finite steps.

- **Aperiodicity**. The *period* of a configuration $\{x\}$ is given by the greatest common divisor of possible numbers of steps to come back to itself. When the period is 1 for all configurations, the Markov chain is called aperiodic.

- **Detailed balance condition**. The transition probability T satisfies $e^{-S[\{x\}]}T[\{x\} \to \{x'\}] = e^{-S[\{x'\}]}T[\{x'\} \to \{x\}]$.

Then, the probability distribution of $\{x^{(k)}\}(k = 1, 2, \cdots)$ converges to $P(x_1, x_2, \cdots, x_p) = e^{-S(x_1, x_2, \cdots, x_p)}/Z$ as the chain becomes longer. The expectation values are obtained by taking the average over the configurations,

$$\langle \hat{O} \rangle = \int dx_1 \cdots dx_p O(x_1, \cdots, x_p) P(x_1, x_2, \cdots, x_p) = \lim_{n \to \infty} \frac{1}{n} \sum_{k=1}^{n} O(x_1^{(k)}, \cdots, x_p^{(k)}). \quad (3)$$

Note that this is not an approximation; *this is exact*. Practically we can have only finitely many configurations, so we can only approximate the right hand side by a finite sum. However there is a systematic way to improve it to arbitrary precision: just make the chain longer.

Although a proof is rather involved, the importance of each condition can easily be understood. Probably the most nontrivial condition for most readers is the detailed balance. Suppose the chain converged to a certain distribution $P[\{x\}]$. Then it has to be 'stationary', or equivalently, it should be invariant when shifted one step:

$$\sum_{\{x\}} P[\{x\}]T[\{x\} \to \{x'\}] = P[\{x'\}] \quad (4)$$

If $P[\{x\}] \propto e^{-S[\{x\}]}$, it follows from the detailed balance condition as

$$\begin{aligned}
\sum_{\{x\}} P[\{x\}]T[\{x\} \to \{x'\}] &= \sum_{\{x\}} P[\{x'\}]T[\{x'\} \to \{x\}] \\
&= P[\{x'\}] \sum_{\{x\}} T[\{x'\} \to \{x\}]
\end{aligned}$$

$$= P[\{x'\}]. \tag{5}$$

I recommend you to follow a complete proof once by looking at an appropriate textbook, but you don't have to keep it in your brain. You will need it only when you try to invent something better than MCMC.

Note that, even if you use exactly the same simulation code, if you take different initial condition or use different sequence of random numbers, you get different chain. Still, the chain always converges to the same statistical distribution.

2.1   Off-topic: Bayesian analysis

MCMC is powerful outside physics as well. To see a little bit of flavor, let us consider the Bayes's theorem,

$$P(B_i|A) = \frac{P(A|B_i)P(B_i)}{\sum_j P(A|B_j)P(B_j)}. \tag{6}$$

Here $P(A|B)$ is the conditional probability: Probability that A is true when the condition B is satisfied. For example $B_1, B_2, B_3 \cdots$ are physicists, high tech engineers, MLB players etc, and A is millionaires.

Suppose $P(A|B_i)$ and $P(B_i)$ are given (e. g. $P(A|B_1) = 10^{-4}, P(A|B_2) = 0.05, P(A|B_3) = 0.8, \cdots$), and we want to derive $P(B_i|A)$. We can identify B_j and $P(A|B_j)P(B_j)$ with the value of the field ϕ, the path integral weight $e^{-S[\phi]}[d\phi]$. The denominator $\sum_j P(A|B_j)P(B_j) = P(A)$ is regarded as the partition function Z. Then we can use MCMC to obtain $P(B_i|A) \sim \frac{e^{-S[\phi]}[d\phi]}{Z}$ via the Bayes's theorem; namely we can collect many samples and see the distribution.

Also if we know $f(B_i) \equiv P(C|B_i)$ you can calculate $P(C|A)$ as

$$P(C|A) = \langle f \rangle. \tag{7}$$

For example C is nice muscle and $P(C|B_1) = P(C|B_2) = 0.01, P(C|B_3) = 0.99, \cdots$.

3   Integration of one-variable functions with Metropolis algorithm

Let us start with the integration of a one-variable function with the Metropolis algorithm [12]. In particular, we will consider the simplest example we can imagine: the Gaussian integral, $S(x) = x^2/2$. This very basic example contains essentially all important ingredients; all other cases are, ultimately, just technical improvements of this example.

Of course we can handle the Gaussian integral analytically. Also there is a much better algorithm for generating Gaussian random numbers (see Appendix B). We use it just for an educational purpose.

3.1 Metropolis Algorithm

Let us consider the weight $e^{-S(x)}$, where $S(x)$ is a continuous function of $x \in \mathbb{R}$ bounded from below. We further assume that $\int e^{-S(x)}dx$ is finite. The Metropolis algorithm gives us a chain of configurations (or just 'values' in the case of single variable) $x^{(0)} \to x^{(1)} \to x^{(2)} \to \cdots$ which satisfies the conditions listed above:

1. Randomly choose $\Delta x \in \mathbb{R}$, and shift $x^{(k)}$ as $x^{(k)} \to x' \equiv x^{(k)} + \Delta x$. ($\Delta x$ and $-\Delta x$ must appear with the same probability, so that the detailed balance condition is satisfied. Here we use the uniform random number between $\pm c$, where $c > 0$ is the 'step size'.)

2. Metropolis test: Generate a uniform random number r between 0 and 1. If $r < e^{-\Delta S}$, where $\Delta S = S(x') - S(x^{(k)})$, then $x^{(k+1)} = x'$, i.e. the new value is 'accepted.' Otherwise $x^{(k+1)} = x^{(k)}$, i.e. the new value is 'rejected.'

3. Repeat the same for $k+1, k+2, \cdots$.

It is an easy exercise to see that all conditions explained above are satisfied:

- It is a Markov Chain, because the past history is not referred either for the selection of Δx or the Metropolis test.

- It is irreducible; for example, any x and x', by taking n large we can make $\frac{x-x'}{n}$ to be in $[-c, c]$, and there is a nonzero probability that $\Delta x = \frac{x-x'}{n}$ appears n times in a row and passes the Metropolis test every time.

- For any x, there is a nonzero probability of $\Delta x = 0$. Hence the period is one for any x.

- If $|x - x'| > c$, $T[x \to x'] = T[x' \to x] = 0$. When $|x - x'| \leq c$, both $\Delta x = x' - x$ and $\Delta x' = x - x'$ are chosen with probability $\frac{1}{2c}$. Let us assume $\Delta S = S[x'] - S[x] > 0$, without a loss of generality. Then the change $x \to x'$ passes the Metropolis test with probability $e^{-\Delta S}$, while $x' \to x$ is always accepted. Hence $T[x \to x'] = \frac{e^{-\Delta S}}{2c}$ and $T[x' \to x] = \frac{1}{2c}$, and $e^{-S[x]}T[x \to x'] = e^{-S[x']}T[x' \to x] = \frac{e^{-S[x']}}{2c}$.

3.1.1 How it works

Let me show a sample code written in C:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
```

```c
int main(void){
  int iter,niter=100;
  int naccept;
  double step_size=0.5e0;
  double x,backup_x,dx;
  double action_init, action_fin;
  double metropolis;

  srand((unsigned)time(NULL));

  /*******************************/
  /* Set the initial configuration */
  /*******************************/
  x=0e0;
  naccept=0;
  /*************/
  /* Main loop */
  /*************/
  for(iter=1;iter<niter+1;iter++){
    backup_x=x;
    action_init=0.5e0*x*x;

    dx = (double)rand()/RAND_MAX;
    dx=(dx-0.5e0)*step_size*2e0;
    x=x+dx;

    action_fin=0.5e0*x*x;
    /******************/
    /* Metropolis test */
    /******************/
    metropolis = (double)rand()/RAND_MAX;
    if(exp(action_init-action_fin) > metropolis)
      /* accept */
      naccept=naccept+1;
    else
      /* reject */
      x=backup_x;
    /**************/
    /* data output */
    /**************/
    printf("%f\n",x);}
}
```

Let me explain the code line by line. Firstly, by

```
srand((unsigned)time(NULL));
```

the seed for the random number generator is set. A default random number generator is used, by using the system clock time to set the seed randomly. For more serious simulations, it is better to use a good generator, say the Mersenne twister.

Then we specify an initial configuration; here we took $x^{(0)} = 0$. **naccept** counts how many times the new values are accepted.

```
x=0e0;
naccept=0;
```

Then we move on to the main part of the simulation, which is inside the following loop:

```
for(iter=1;iter<niter+1;iter++){ .... }
```

Here, **iter** corresponds to k, and **niter** is the number of configurations we will collect during the simulation.

Inside the loop, the first thing we have to do is to save the value of $\mathbf{x} = x^{(k)}$, because it may or may not be updated:

```
backup_x=x;
```

Then **action_init**$= S(x^{(k)})$ is calculated.

Now we have to generate a random variation $\mathbf{dx} = \Delta x$ with an appropriate step size, and shift x to $x' = x^{(k)} + \Delta x$. We can generate a uniform random number in $[0, 1]$ by **rand()/RAND_MAX**. From this we can easily get $-c < \Delta x < c$.

```
dx = (double)rand()/RAND_MAX;
dx=(dx-0.5e0)*step_size*2e0;
x=x+dx;
```

By using x', **action_fin**$= S(x')$ is calculated. Note that $\mathbf{x}$ and $\mathbf{backup\_x}$ in the code correspond to x' and $x^{(k)}$.

Finally we perform the Metropolis test:

```
metropolis = (double)rand()/RAND_MAX;
if(exp(action_init-action_fin) > metropolis)
  /* accept */
  naccept=naccept+1;
else
  /* reject */
  x=backup_x;
```

10

metropolis is a uniform random number in $[0, 1]$, which corresponds to r. Depending on the result of the test, we accept or reject x'.

We emphasize again that *all MCMC simulations have exactly the same structure*; there are many fancy algorithms, but essentially, they are all about improving the step $x \rightarrow x + \Delta x$.

We take $x^{(0)} = 0$, and Δx to be a uniform random number between -0.5 and 0.5. (As we will see later, this parameter choice is not optimal.) In Fig. 2, we show the distribution of $x^{(1)}, x^{(2)}, \cdots, x^{(n)}$, for $n = 10^3, 10^5$ and 10^7. We can see that the distribution converges to $e^{-x^2/2}/\sqrt{2\pi}$. The expectation values $\langle x \rangle = \frac{1}{n} \sum_{k=1}^{n} x^{(k)}$ and $\langle x^2 \rangle = \frac{1}{n} \sum_{k=1}^{n} \left(x^{(k)} \right)^2$ are plotted in Fig. 3. As n becomes large, they converge to the right values, 0 and 1.

Note that the step size c should be chosen so that the acceptance rate is not too high, not too low. If c is too large, the acceptance rate becomes extremely low, then the value is rarely updated. If c is too small, the acceptance rate is almost 1, but the change of the value at each step is extremely small. In both cases, huge amount of configurations are needed in order to approximate the integration measure accurately. The readers can confirm it by changing the step size in the sample code. (We will demonstrate it in Sec. 3.2.)

Typically the acceptance rate 30% – 80% is good. But it can heavily depend on the detail of the system and algorithm; See Sec. 3.2.2 and Sec. 4.2.3 for details.
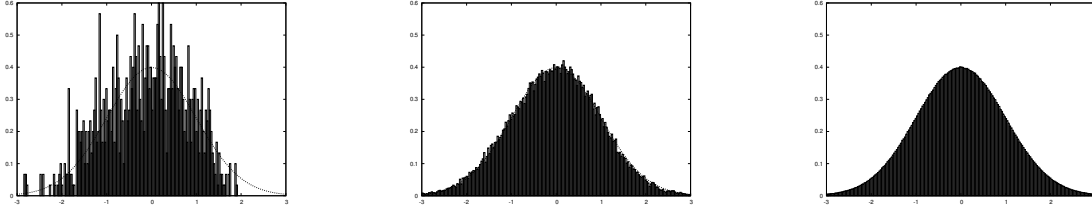

Figure 2: The distribution of $x^{(1)}, x^{(2)}, \cdots, x^{(n)}$, for $n = 10^3, 10^5$ and 10^7, and $\frac{e^{-x^2/2}}{\sqrt{2\pi}}$.
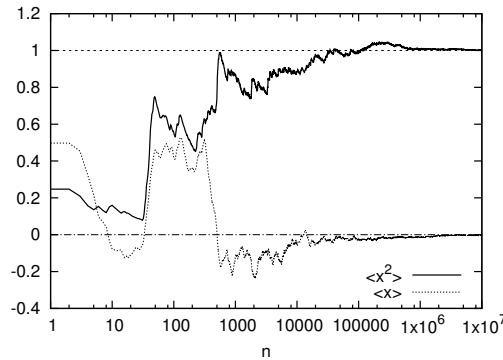

Figure 3: $\langle x \rangle = \frac{1}{n} \sum_{k=1}^{n} x^{(k)}$ and $\langle x^2 \rangle = \frac{1}{n} \sum_{k=1}^{n} \left(x^{(k)} \right)^2$. As n becomes large, they converge to the right values, 0 and 1.

A bad example

It is instructive to see wrong examples. Let us take Δx from $\left[-\frac{1}{2}, 1\right]$, so that the detailed balance condition is violated; for example $0 \to 1$ has a finite probability while $1 \to 0$ is impossible. Then, as we can see from Fig 4, the chain does not converge to the right probability distribution.
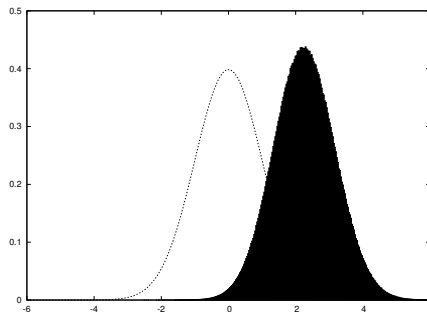

Figure 4: The distribution of $x^{(1)}, x^{(2)}, \cdots, x^{(n)}$ for $n = 10^7$, with *wrong* algorithm with $\Delta x \in \left[-\frac{1}{2}, 1\right]$. The dotted line is the right Gaussian distribution $\frac{e^{-x^2/2}}{\sqrt{2\pi}}$.

3.2   Autocorrelation and Thermalization

In MCMC, $x^{(k+1)}$ is obtained from $x^{(k)}$. In general, they are correlated. This correlation is called *autocorrelation*. The autocorrelation can exist over many steps. The autocorrelation length depends on the detail of the theory, algorithm and the parameter choice. Because of the autocorrelation, some cares are needed.

In the above, we have set $x^{(0)} = 0$, because we knew it is 'the most important configuration'. What happens if we start with an atypical value, say $x^{(0)} = 100$? It takes some time for typical values to appear, due to the autocorrelation. The history of the Monte Carlo simulation with this initial condition is shown in Fig. 5. The value of x eventually reaches to 'typical values' $|x| \lesssim 1$ — we often say 'the configurations are thermalized' (note that the same term 'thermalized' has another meaning as well, as we will see shortly) —, but a lot of steps are needed. If we include 'unthermalized' configurations when we estimate the expectation values, we will suffer from huge error unless the number of configurations are extremely large. We should discard unthermalized configurations, say $n \lesssim 1000$.[3]

In generic, more complicated situation, we don't a priori know what the typical configurations look like. Still, whether the configurations are thermalized or not can be seen by looking at several observables. As long as they are changing monotonically, it is plausible that the configuration is moving toward a typical one. When they start to oscillate around

---

[3] Number of steps needed for the thermalization is sometimes called 'burn-in time' or 'mixing time'.

certain values (see Fig. 6, we can see a fluctuation around $x = 0$), it is reasonable to think the configuration has been thermalized.[4]

Fig. 6 is a zoom-up of Fig. 5, from $n = 1000$ to $n = 2000$. We can see that the values of x can be strongly correlated unless they are 20 or 30 step separated. (We will give a quantitative analysis in Sec. 3.2.1.) When we estimate the statistical error, we should not treat all configuration as independent; rather we have only 1 independent configuration every 20 or 30 steps. Note also that we need sufficiently many independent configurations in order to estimate the expectation values reliably. We often say the simulation has been thermalized when we have sufficiently many independent configurations so that the expectation values are stabilized.
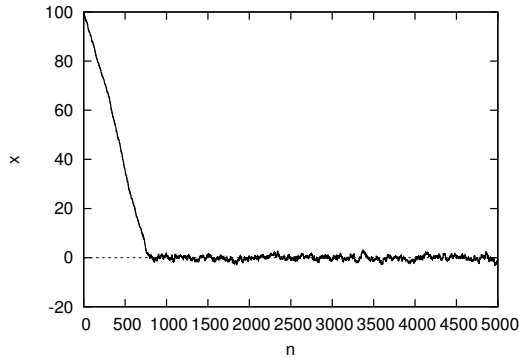
Figure 5: Monte Calro history of the Gaussian integral with the Metropolis algorithm, $\Delta x \in [-0.5, 0.5]$. We took the initial value to be a very atypical value, $x = 100$. It takes a lot of steps to reach typical values $|x| \sim 1$.

3.2.1 Jackknife method

The Jackknife method provides us with a simple way to estimate the autocorrelation length. Here we assume the quantity of interest can be calculated for each sample.[5] For more generic cases, see Appendix C.

In the Jackknife method, we first divide the configurations to bins with width w; the first bin consists of $\{x^{(1)}\}, \{x^{(2)}\}, \cdots, \{x^{(w)}\}$, the second bin is $\{x^{(w+1)}\}, \{x^{(w+2)}\}, \cdots, \{x^{(2w)}\}$, and so on. Suppose we have n bins. Then we define the average of an observable $f(x)$ with k-th bin removed,

$$\overline{f}^{(k,w)} \equiv \frac{1}{(n-1)w} \sum_{j \notin k\text{-th bin}} f(x^{(j)}). \tag{8}$$

---

[4]For more careful analysis, we can vary the number of configurations removed and take it large enough so that the average values do not change any more.

[5]The correlation function is in this class. The mass of particle excitation is not; we need to calculate two-point function by using many samples and then extract the mass from its exponential decay.

13
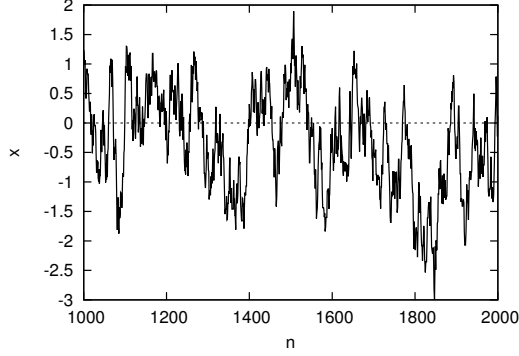
Figure 6: A zoom-up of Fig. 5, from $n = 1000$ to $n = 2000$. The values of x can be strongly correlated unless they are at least 20 or 30 steps separated.

The average value

$$\overline{f} \equiv \frac{1}{n} \sum_k \overline{f}^{(k,w)} \tag{9}$$

is the same as the average of all samples, $\frac{1}{nw} \sum_j f(x^{(j)})$, for the class of quantities we are discussing. The *Jackknife error* is defined by

$$\Delta_w \equiv \sqrt{\frac{n-1}{n} \sum_k \left(\overline{f}^{(k,w)} - \overline{f} \right)^2}. \tag{10}$$

By using

$$\tilde{f}^{(k,w)} \equiv \frac{1}{w} \sum_{j \, \in \, k\text{-th bin}} f(x^{(j)}), \tag{11}$$

we can easily see

$$\overline{f}^{(k,w)} - \overline{f} = \frac{\overline{f} - \tilde{f}^{(k,w)}}{n-1}. \tag{12}$$

Hence

$$\Delta_w \equiv \sqrt{\frac{1}{n(n-1)} \sum_k \left(\tilde{f}^{(k,w)} - \overline{f} \right)^2}. \tag{13}$$

Namely Δ_w is the standard error obtained by treating $\tilde{f}^{(k,w)}$ to be independent samples.

Typically, as w becomes large, Δ_w increases and then becomes almost constant at certain value of w, which we denote by w_c. This w_c and Δ_{w_c} give good estimates of the autocorrelation length and the error bar.

14

It can be understood as follows. Let us consider two bin sizes w and $2w$. Then

$$\tilde{f}^{(k,2w)} = \frac{\tilde{f}^{(2k-1,w)} + \tilde{f}^{(2k,w)}}{2}, \tag{14}$$

$$
\begin{aligned}
\Delta_{2w} &= \sqrt{\frac{1}{\frac{n}{2}\left(\frac{n}{2}-1\right)} \sum_{k=1}^{n/2} \left(\tilde{f}^{(k,2w)} - \overline{f}\right)^2} \\
&= \sqrt{\frac{4}{n(n-2)} \sum_{k=1}^{n/2} \left(\frac{\left(\tilde{f}^{(2k-1,w)} - \overline{f}\right)}{2} + \frac{\left(\tilde{f}^{(2k,w)} - \overline{f}\right)}{2}\right)^2}. \tag{15}
\end{aligned}
$$

If w is sufficiently large, $\tilde{f}^{(2k-1,w)} - \overline{f}$ and $\tilde{f}^{(2k,w)} - \overline{f}$ should be independent, and the cross-term $\left(\tilde{f}^{(2k-1,w)} - \overline{f}\right) \cdot \left(\tilde{f}^{(2k,w)} - \overline{f}\right)$ should average to zero after summing up with respect to sufficiently many k. Then

$$\Delta_{2w} \sim \sqrt{\frac{1}{n^2} \sum_{k=1}^{n} \left(\tilde{f}^{(k,w)} - \overline{f}\right)^2} \sim \Delta_w. \tag{16}$$

In this way, Δ_w becomes approximately constant when w is large enough so that $\tilde{f}^{(k,w)}$ can be treated as independent samples. (Note that n must also be large for the above estimate to hold.) Δ_w is the standard error of these 'independent samples'.

In Fig. 7, $\langle x^2 \rangle$ and Jackknife error Δ_w are shown by using first 50000 samples. We can see that $w_c = 50$ is a reasonably safe choice; $w_c = 20$ is already in the right ballpark. In Fig. 8, bin-averaged values with $w = 50$ are plotted. They do look independent. We obtained $\langle x^2 \rangle = 0.982 \pm 0.012$, which agree reasonably well with the analytic answer, $\langle x^2 \rangle = 1$.
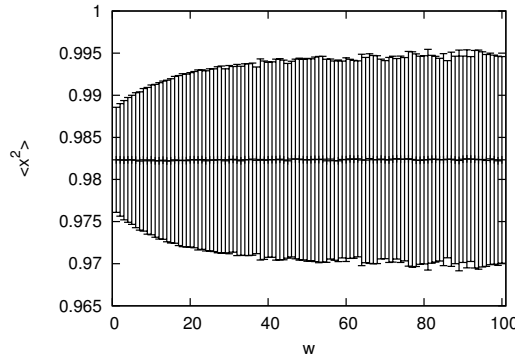
Figure 7: $\langle x^2 \rangle$ and Jackknife error Δ_w with 50000 samples.

15
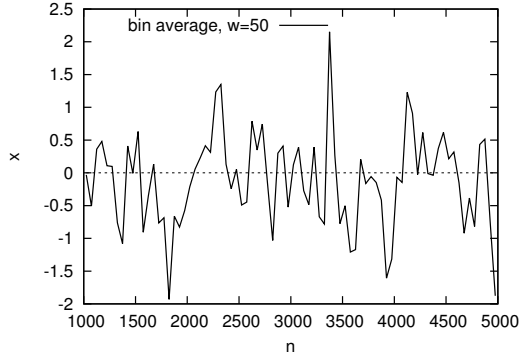
Figure 8: Bin-averaged version of Fig. 6, with bigger window for n, with $w = 50$.

step size c	acceptance	$c \times$ acceptance
0.5	0.9077	0.454
1.0	0.8098	0.810
2.0	0.6281	1.256
3.0	0.4864	1.459
4.0	0.3911	1.564
6.0	0.2643	1.586
8.0	0.1993	1.594

Table 1: Step size vs acceptance rate, total 10000 samples.

3.2.2 Tuning the simulation parameters

In order to run the simulation efficiently, we should tune parameters so that we can obtain *more independent samples with less cost.*[6]

In the current example (Gaussian integral with uniform random number), when the step size c is too large, unless $\Delta x \lesssim 1$ the configuration is rarely updated; the acceptance rate and the autocorrelation length scale as $1/c$ and c, respectively. On the other hand, when c is too small, the configurations are almost always updated, but only tiny amount. This is just a random walk with a step size c, and hence the average change after n steps is $c\sqrt{n}$. Therefore the autocorrelation length should scale as $n \sim 1/c^2$. We expect the autocorrelation is minimized between these two regions. In Table 1, we have listed the acceptance rate for several values of c. We can see that the large-c scaling ($c \times$ acceptance $\sim$ const) sets in at around $c = 2 \sim c = 4$. In Fig. 9 we have shown how $\langle x^2 \rangle$ converges to 1 as the number of configurations increases. We can actually see that $c = 2$ and $c = 4$ show faster convergence compared to too small or too large c.

---

[6]In parallelized simulations, the notion of the *cost* is more nontrivial because time is money. Sometimes you may want to invest more electricity and machine resources to obtain the same result with shorter time.
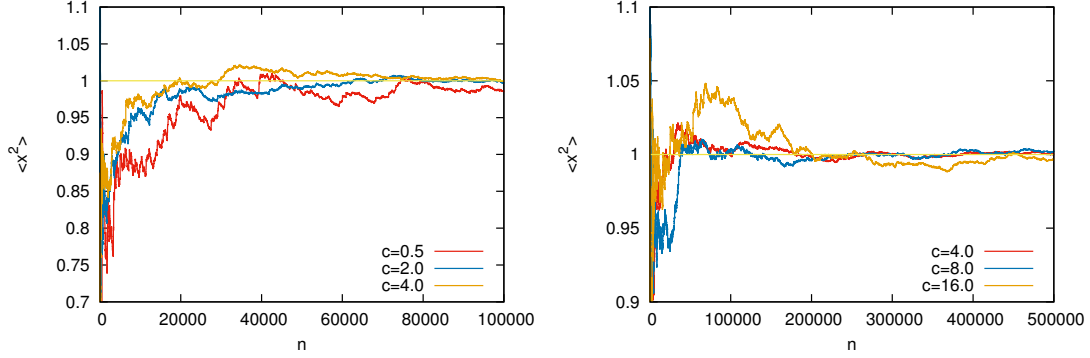
Figure 9: $\langle x^2 \rangle = \frac{1}{n} \sum_{k=1}^n \left(x^{(k)} \right)^2$ for several different step sizes c.

3.3 How to calculate partition function

In MCMC, we cannot directly calculate the partition function Z; we can only see the expectation values. Usually the partition function is merely a normalization factor of the path integral measure which does not affect the path integral, so we do not care. But sometimes it has interesting physical meanings; for example it can be used to test the conjectured dualities between supersymmetric theories.

Suppose you want to calculate $Z = \int dx e^{-S(x)}$, where $S(x)$ is much more complicated than $S_0(x) = x^2/2$. By using MCMC, we can calculate the ratio between Z and $Z_0 = \int dx e^{-S_0(x)} = \sqrt{2\pi}$,

$$\frac{Z}{Z_0} = \frac{1}{Z_0} \int dx e^{-S_0} \cdot e^{S_0 - S} = \left\langle e^{S_0 - S} \right\rangle_0, \tag{17}$$

where $\langle \cdot \rangle_0$ stands for the expectation value with respect to the action S_0. Because we know Z_0 analytically, we can determine Z.

3.3.1 Overlapping problem and its cure

The method described above can always work *in principle*. In practice, however, it fails when the probability distributions $\rho(x) = \frac{e^{-S(x)}}{Z}$ and $\rho_0(x) = \frac{e^{-S_0(x)}}{Z_0}$ do not have sufficiently large overlap. As a simple example, let us consider $S = (x - c)^2/2$ (though you can analytically handle it!). Then $\rho(x)$ and $\rho_0(x)$ have peaks around $x = c$ and $x = 0$, respectively. When c is very large, say $c = 100$, the value of $e^{S_0 - S}$ appearing in the simulation is almost always an extremely small number $\sim e^{-5000}$, and once every e^{+5000} steps or so we get an extremely large number $\sim e^{+5000}$. And they average to $\frac{Z}{Z_0} = 1$. Clearly, we cannot get an accurate number if we truncate the sum at a realistic number of configurations. It happens because of the absence of the overlap of $\rho(x)$ and $\rho_0(x)$, or equivalently, because important configurations in two different theories are different;

hence the 'operator' e^{S_0-S} behaves badly at the tail of $\rho_0(x)$. This is so-called *overlapping problem*.[7]

In the current situation, the overlapping problem can easily be solved as follows. Let us introduce a series of actions S_0, S_1, S_2, ..., $S_k = S$. We choose them so that S_i and S_{i+1} are sufficiently close and the ratio $\frac{Z_{i+1}}{Z_i}$, where $Z_i = \int dx e^{-S_i(x)}$, can be calculated without the overlapping problem. For example we can take $S_i = \frac{1}{2}\left(x - \frac{i}{k}c\right)^2$ with $\frac{c}{k} \sim 1$. Then we can obtain $Z = Z_k$ by calculating $\frac{Z_1}{Z_0}$, $\frac{Z_2}{Z_1}$, $\cdots$, $\frac{Z_k}{Z_{k-1}}$. The same method can be applied to any complicated $S(x)$, as long as $e^{-S(x)}$ is real and positive.

This rather primitive method is actually powerful; for example the partition function of ABJM theory [13] at finite coupling and finite N has been calculated accurately by using this method [4].

3.4   Common mistakes

Let us see some common mistakes below.

3.4.1   Don't change step size during the run

Imagine the probability distribution you want to study has a bottleneck like in Fig. 10. For example if $S(x) = -\log\left(e^{-\frac{x^2}{2}} + e^{-\frac{(x-100)^2}{2}}\right)$ then $e^{-S(x)}$ is strongly suppressed between two peaks at $x = 0$ and $x = 100$. By using a small step size $c \sim 1$ you can sample one of the peaks efficiently, but then the other peak cannot be sampled. Then in order to go across the bottleneck you would be tempted to change the step size c when you come close to the bottleneck. You would want to make the step size larger so that you can jump over the bottle neck, or you would want to make the step size smaller so that you can slowly penetrate into the bottle neck. But if you do so, you obtain a wrong result, because the transition probability can depend on the past history. *You must not change the step size during the simulation.*

But it does not mean that you cannot use multiple fixed step sizes; it is allowed to change the step size if the conditions listed in Sec. 2 are not violated. For example we can take $c = 1$ for odd steps and $c = 100$ for even steps; see Fig. 11.

Or we can throw a dice, namely randomly choose step size $c = 1, 2, 3, 4, 5, 6$ with probability 1/6. As long as the conditions listed in Sec. 2, in particular the detailed balance, are not violated, you can do whatever you want.

Similar temptation of evil is common in muilti-variable case. For example in the lattice gauge theory simulation it often happens that the acceptance is extremely low until the system thermalizes. Then we can use smaller step size just to make the system thermalize, and then start actual data-taking with a larger step size.[8]   Or it occasionally happens

---

[7]In SYM, the overlapping problem can appear combined with the *sign problem*; we will revisit this point in Sec. 7.2.

[8]Another common strategy to reach the thermalization is to turn off the Metropolis test.
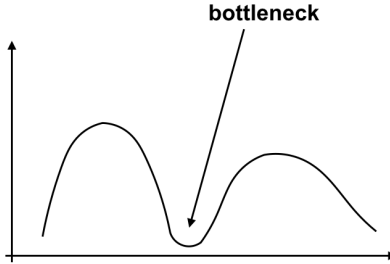
Figure 10: If the probability distribution has a bottleneck, the acceptance rate goes down there.
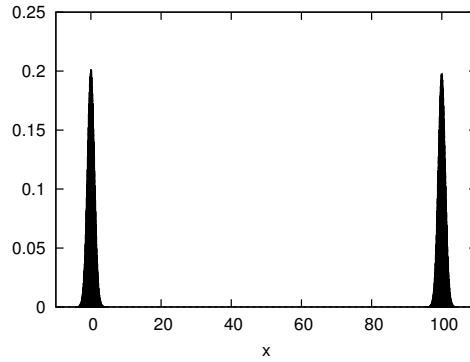
Figure 11: A histogram for $S(x) = -\log\left(e^{-\frac{x^2}{2}} + e^{-\frac{(x-100)^2}{2}}\right)$ with Metropolis, step size $c = 1$ for odd steps and $c = 100$ for even steps, 10^7 samples. The solid curve (which is actually invisible because it agrees with the histogram too precisely...) is the exact answer, $\frac{e^{-\frac{x^2}{2}} + e^{-\frac{(x-100)^2}{2}}}{2\sqrt{2}}$.

that the simulation is trapped at a rare configuration so that the acceptance rate becomes almost zero. In such case, it would be useful to use multiple step sizes (the ordinary and very small).

3.4.2   Don't mix independent simulations with different step sizes

This is similar to Sec. 3.4.1: *when you have several independent runs with different step sizes, you must not mix them to evaluate the expectation value*, unless you pay extra cares for the error analysis. Although each stream are guaranteed to converge to the same distribution, if you truncate them at finite number of configurations each stream contains different uncontrollable systematic error.

Note however that, if you can estimate the autocorrelation time of each stream reliably (for that each stream has to be sufficiently long), you can mix different streams with proper weights, with a careful error analysis.

3.4.3   Make sure that random numbers are really random

In actual simulations, random numbers are not really random, they are just pseudo-random. But we have to make sure that they are sufficiently random. In Fig.12, we used the same sequence of random numbers repeatedly every 1000 steps. The answer is clearly wrong.

This mistake is very common;[9] when one simulates a large system, it will take days or months, so one has to split the simulation to small number of steps. Then when one submits a new job by mistake one would use the same 'random numbers' again, for example by resetting the seed of random numbers to the same number.

Figure 12: Gaussian integral with Metropolis, $c = 1$, with *non-random* numbers; we chose the same random number sequence every 1000 steps. $\langle x^2 \rangle = \frac{1}{n} \sum_{k=1}^{n} \left(x^{(k)} \right)^2$ converges to a wrong number, which is different from 1.

---

[9]Yes, I did.

3.4.4 Remark when you use Mathematica for larger scale simulations

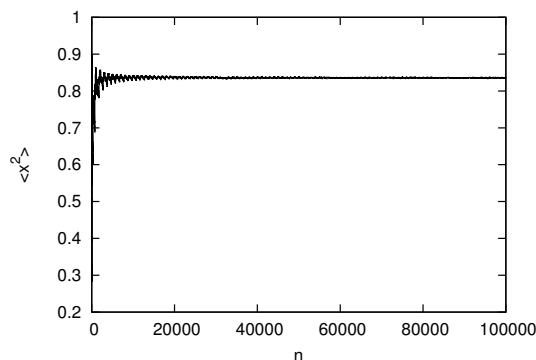I saw several people tried to use Mathematica for MCMC of systems of moderate size $(O(10^3) - O(10^4))$ variables) and could not run the codes except when the system size is small. Probably the problem was that unless one understands Mathematica well one can unintentionally use nice features such as symbolic calculations which are not needed for such primitive calculations used in MCMC. The same remark can apply to other advanced softwares as well.

The simplest solution to this problem is to keep advanced softwares for advanced tasks, and avoid using them for such simple things like MCMC. Unless you use a rare special function or something, it is unlikely that you need anything more than C or Fortran.

But by using those softwares you may be able to save the time for coding. As long as the system size you want to study is small, you don't have to to worry; 10 seconds and 10 minutes are not that different and you will spend more time for the coding anyways. When you want to do heavier calculations, make sure to understand the software well and avoid using unnecessary features.

3.5 Sign problem

So far we have assumed $e^{-S(x)} \geq 0$. This was necessary because we interpreted $e^{-S(x)}$ as a 'probability'. But in physics we often encounter $e^{-S(x)} < 0$, or sometimes $e^{-S(x)}$ can be complex. Then a naive MCMC approach does not work. This is infamous *sign problem* (or *phase problem*, when $e^{-S(x)}$ is complex). Although no generic solution of the sign problem is known, there are various theory-specific solutions. We will come back to this point in Sec. 7.2.2.

3.6 What else do we need for lattice gauge theory simulations?

The advantage of the Metropolis algorithm is clear: it is simple. It is extremely simple and applicable to any theory, as long as the 'sign problem' does not exist. When it works, just use it. For example, for simple matrix model calculations like the one in [2],[10] you don't need anything more than Metropolis and your laptop.

But our budget is limited and we cannot live forever. So sometimes we have to reduce the cost and make simulations faster. We should use better algorithms, better lattice actions

---

[10]In that paper, we studied the symmetry breaking in Twisted Eguchi-Kawai model at large N. In 1980's people used the best computers available and studied $N \lesssim 16$. They did not observe a symmetry breaking. In 2006, I wrote a Metropolis code spending an hour or so, and studied $25 \lesssim N \lesssim 100$ with my laptop. Within a few hours I could see a clear signature of the symmetry breaking. In order to understand the detail of the symmetry breaking pattern we had to study many parameters, so we ran the same code on a cluster machine. One of my collaborators was serious enough to write a more sophisticated code to go to much larger N, with which we could study $N \gtrsim 100$. Note that it is a story from 2006 to 2007; now you can do much better job with Metropolis and your laptop.

and better observables, which are 'better' in the following sense:

- The autocorrelation length is shorter.

- Easier to parallelize. In lattice gauge theory, it typically means that we should utilize the sparseness of the Dirac operator.

- Find good observables and good measurement methods which are easier to calculate, have less statistical fluctuations, and/or show faster convergence to the continuum limit.

For quantum field theories, especially when the fermions are involved, HMC is effective. RHMC is a variant of HMC which is applicable to SYM.

4 Integration of multiple variables and bosonic QFT

Once a regularization is given, the path-integral is merely an integral with multiple variables. Hence let us start with a simple case of a matrix integral, then proceed to QFT.

4.1 Metropolis for multiple variables

Generalization of the Metropolis algorithm (Sec. 3.1) to multiple variables $(x_1, x_2, \cdots, x_p)$ is straightforward. For example, we can do as follows:

1. For all $i = 1, 2, \cdots, p$, randomly choose $\Delta x_i \in [-c_i, +c_i]$, and shift $x_i^{(k)}$ as $x_i^{(k)} \to x_i' \equiv x_i^{(k)} + \Delta x_i$. Note that the step size c_i can be different for different x_i.

2. Metropolis test: Generate a uniform random number r between 0 and 1. If $r < e^{S[x^{(k)}] - S[x']}$, $\{x^{(k+1)}\} = \{x'\}$, i.e. the new configuration is 'accepted.' Otherwise $\{x^{(k+1)}\} = \{x^{(k)}\}$, i.e. the new configuration is 'rejected.'

3. Repeat the same for $k + 1, k + 2, \cdots$.

One can also do as follows:

1. Randomly choose $\Delta x_1 \in [-c_1, +c_1]$, and shift $x_1^{(k)}$ as $x_1^{(k)} \to x_1' \equiv x_1^{(k)} + \Delta x_1$.

2. Metropolis test: Generate a uniform random number r between 0 and 1. If $r < e^{S[x^{(k)}] - S[x']}$, $x_1^{(k+1)} = x_1'$, i.e. the new value is 'accepted.' Otherwise $x_1^{(k+1)} = x_1^{(k)}$, i.e. the new configuration is 'rejected.' For other values of i we don't do anything, namely $x_i^{(k+1)} = x_i^{(k)}$ for $i = 2, 3, \cdots, p$.

3. Repeat the same for $i = 2, 3, \cdots, p$.

4. Repeat the same for $k + 1, k + 2, \cdots$.

4.1.1 How it works

Let us consider a one matrix model,

$$S[\phi] = N\mathrm{Tr}\left(\frac{1}{2}\phi^2 + V(\phi)\right),\tag{18}$$

where ϕ is an $N \times N$ Hermitian matrix, $\phi_{ji} = \phi_{ij}^*$. The potential $V(\phi)$ can be anything as long as the partition function is convergent; say $V(\phi) = \phi^4$.

The code has exactly the same structure as the sample code in Sec. 3.1; we should calculate $S[\phi]$ instead of the Gaussian weight, and instead of x we can shift ϕ by using N^2 real random numbers.

As N gets larger, more and more portion of the integral region becomes unimportant. Therefore, if we vary all the components simultaneously, ΔS is typically large and the acceptance rate is very small, unless we take the step size to be small. To avoid it, we can vary one component at each time. Note that, when only ϕ_{ij} and $\phi_{ji} = \phi_{ij}^*$ are varied, one should save the computational cost by calculating ϕ_{ij}-dependent part instead of $S[\phi]$ itself; the latter costs $O(N^3)$, though the former costs only $O(N^2)$.

4.2 Hybrid Monte Carlo (HMC) Algorithm

The important configurations are like bottom of a valley; the altitude is the value of the action. This is a valley in the phase space, whose dimension is very large. So if the configuration is literally randomly varied, like in the Metropolis algorithm, the action almost always increases a lot. Hence with the Metropolis algorithm the acceptance rate is small unless the step size is extremely small, and it causes rather long autocorrelation length. The Hybrid Monte Carlo (HMC) algorithm [14] avoids the problem of a long autocorrelation by effectively crawling along the bottom of the valley; this is a 'hybrid' of molecular dynamical method and Metropolis algorithm.

In HMC algorithm, sets of configurations $\{x^{(k)}\}$ $(k = 0, 1, 2, \cdots)$ are generated in the following manner. Firstly, $\{x^{(0)}\}$ can be arbitrary. Once $\{x^{(k)}\}$ is obtained, $\{x^{(k+1)}\}$ is obtained as follows.

1. Randomly generate auxiliary momenta $P_i^{(k)}$, which are 'conjugate' to $x_i^{(k)}$, with probabilities $\frac{1}{\sqrt{2\pi}}e^{-(P_i^{(k)})^2/2}$. To generate Gaussian random numbers, the Box-Muller algorithm is convenient; see Appendix B.

2. Calculate the 'Hamiltonian' $H_i = S[x^{(k)}] + \frac{1}{2}\sum_i(P_i^{(k)})^2$.

3. Then we consider 'time evolution' along an auxiliary time τ (which is not the Euclidean time!). We set the initial condition to be $x^{(k)}(\tau = 0) = x^{(k)}$ and $P^{(k)}(\tau = 0) = P^{(k)}$, and use the leap frog method (see below) to calculate $x^{(k)}(\tau_f)$ and $P^{(k)}(\tau_f)$, where τ_f is related to the input parameters $\Delta\tau$ and N_τ by $\tau_f = N_\tau\Delta\tau$.

   This process is called 'molecular evolution.'

4. Calculate $H_f = S[x^{(k)}(\tau_f)] + \frac{1}{2} \sum_i (P_i^{(k)}(\tau_f))^2$.

5. Metropolis test: Generate a uniform random number r between 0 and 1. If $r < e^{H_i - H_f}$, $x^{(k+1)} = x^{(k)}(\tau_f)$, i.e. the new configuration is 'accepted.' Otherwise $x^{(k+1)} = x^{(k)}$, i.e. the new configuration is 'rejected.'

At first sight it is a rather complicated algorithm. Why do we introduce such auxiliary dynamical system? The key is the 'energy conservation'.

Just like Metropolis, the change of the configuration is random due to the randomness of the choice of auxiliary momenta P_i. If we just compared the initial and final values of the action, the change would be equally large. However in HMC the change of the auxiliary Hamiltonian matters in the Metropolis test. (Please accept this fact for the moment, in the next paragraph we will explain the reason.) If we keep $N_\tau \Delta\tau$ fixed and send N_τ to infinity, then the Hamiltonian is exactly conserved and new configurations are always accepted. By taking $N_\tau \Delta\tau$ to be large, new configurations can be substantially different from the old ones. (Of course, calculation cost increase with N_τ. So we have to find a sweet spot, with moderately large N_τ and moderately small $\Delta\tau$.)

To check the detailed balance condition $e^{-S[x]} T[\{x\} \to \{x'\}] = e^{-S[x']} T[\{x'\} \to \{x\}]$, note that the leap-frog method is designed so that the molecular evolution is reversible; namely, if we start with $x^{(k)}(\tau_f)$ and $-P^{(k)}(\tau_f)$, the final configuration is $x^{(k)}(\tau = 0)$ and $-P^{(k)}(\tau = 0)$. Hence, if $\{x, p\}$ evolves to $\{x', p'\}$, then (by assuming $H - H' < 0$ without loss of generality) $e^{-S[x]} T[\{x\} \to \{x'\}] \propto e^{-S[x]} e^{-p^2/2}$, while $e^{-S[x']} T[\{x'\} \to \{x\}] \propto e^{-S[x']} e^{-p'^2/2} e^{-S[x] - p^2/2 + S[x'] + p'^2/2} = e^{-S[x]} e^{-p^2/2}$, with the same proportionality factor.

In fact the HMC algorithm works even when we take different τ for each x_i; see Sec. 4.3. We can use different τ on depending the field, momentum of the mode, etc. The HMC algorithm is powerful especially when we have to deal with fermions, as we will explain later.

4.2.1 Leap frog method

The leap frog method is a clever way to discretize the continuum Hamiltonian equation keeping the reversibility, which is crucial for assuring the detailed balance condition.

The continuum Hamiltonian equation is given by

$$\frac{dp_i}{d\tau} = -\frac{\partial H}{\partial x_i} = -\frac{\partial S}{\partial x_i}, \qquad \frac{dx_i}{d\tau} = \frac{\partial H}{\partial p_i} = p_i. \tag{19}$$

The leap frog method goes as follows[11]:

1. $x_i(\Delta\tau/2) = x_i(0) + p_i(0) \cdot \frac{\Delta\tau}{2}$ (Step 1 in Fig. 13).

---

[11]Be careful about a factor of 1/2 and ordering of the operations. These are crucial for the reversibility of the molecular evolution and the detailed balance condition.

2. For $n = 1, 2, N_\tau - 1$, repeat it:

$p_i(n\Delta\tau) = p_i((n-1)\Delta\tau) - \frac{\partial S}{\partial x_i}((n-1/2)\Delta\tau) \cdot \Delta\tau$ (Step $2, 4, \cdots, 2N_\tau - 2$ in Fig. 13),

then

$x_i((n+1/2)\Delta\tau) = x_i((n-1/2)\Delta\tau) + p_i(n\Delta\tau) \cdot \Delta\tau$ (Step $3, \cdots, 2N_\tau - 1$ in Fig. 13).

3. Finally,

$p_i(N_\tau\Delta\tau) = p_i((N_\tau - 1)\Delta\tau) - \frac{\partial S}{\partial x_i}((N_\tau - 1/2)\Delta\tau) \cdot \Delta\tau$ (Step $2N_\tau$ in Fig. 13),

then

$x_i(N_\tau\Delta\tau) = x_i((N_\tau - 1/2)\Delta\tau) + p_i(N_\tau\Delta\tau) \cdot \frac{\Delta\tau}{2}$ (Step $2N_\tau + 1$ in Fig. 13).
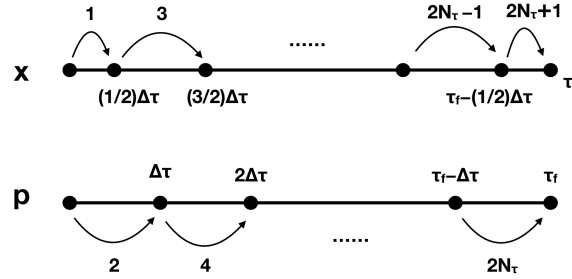
Figure 13: Leap-frog method.

4.2.2   How it works, 1 — Gaussian Integral

As the simplest example, let us go back to the one-variable case and start with the Gaussian integral again.[12] Here is a sample code:

```
#include <iostream>
#include <cmath>
#include<fstream>
const int niter=10000;
const int ntau=40;
const double dtau=1e0;
/********************************************************************/
/*** Gaussian Random Number Generator with Box Muller Algorithm ***/
/********************************************************************/
int BoxMuller(double& p, double& q){
```

---

[12]This is an extremely stupid example, given that we need the Gaussian random number for the HMC algorithm! But I believe it is still instructive.

25

```
  double pi;
  double r,s;
  pi=2e0*asin(1e0);
  //uniform random numbers between 0 and 1
  r = (double)rand()/RAND_MAX;
  s = (double)rand()/RAND_MAX;
  //Gaussian random numbers,
  //with weights proportional to e^{-p^2/2} and e^{-q^2/2}
  p=sqrt(-2e0*log(r))*sin(2e0*pi*s);
  q=sqrt(-2e0*log(r))*cos(2e0*pi*s);

  return 0;
}
/******************************/
/*** Calculation of the action ***/
/******************************/
// When you change the action, you should also change dH/dx,
// specified in "calc_delh".
double calc_action(const double x){

  double action=0.5e0*x*x;

  return action;
}
/***********************************/
/*** Calculation of the Hamiltonian ***/
/***********************************/
double calc_hamiltonian(const double x,const double p){

  double ham;

  ham=calc_action(x);

  ham=ham+0.5e0*p*p;

  return ham;
}
/*************************/
/*** Calculation of dH/Dx ***/
/*************************/
// Derivative of the Hamiltonian with respect to x,
// which is equivalent to the derivative of the action.
```

```cpp
// When you change "calc_action", you have to change this part as well.
double calc_delh(const double x){

  double delh=x;

  return delh;
}
/**************************/
/*** Molecular evolution ***/
/**************************/
int Molecular_Dynamics(double& x,double& ham_init,double& ham_fin){

  double p;
  double delh;
  double r1,r2;

  BoxMuller(r1,r2);
  p=r1;

  //*** calculate Hamiltonian ***
  ham_init=calc_hamiltonian(x,p);
  //*** first step of leap frog ***
  x=x+p*0.5e0*dtau;
  //*** 2nd, ..., Ntau-th steps ***
  for(int step=1; step!=ntau; step++){
    delh=calc_delh(x);
    p=p-delh*dtau;
    x=x+p*dtau;
  }
  //*** last step of leap frog ***
  delh=calc_delh(x);
  p=p-delh*dtau;
  x=x+p*0.5e0*dtau;
  //*** calculate Hamiltonian again ***
  ham_fin=calc_hamiltonian(x,p);

  return 0;
}

int main()
{
  double x;
```

```cpp
  double backup_x;
  double ham_init,ham_fin,metropolis,sum_xx;

  srand((unsigned)time(NULL));
  /********************************/
  /* Set the initial configuration */
  /********************************/
  x=0e0;
  /****************/
  /*** Main part ***/
  /****************/
  std::ofstream outputfile("output.txt");
  int naccept=0;//counter for the number of acceptance
  sum_xx=0e0;//sum of x^2, useed for <x^2>

  for(int iter=0; iter!=niter; iter++){

    backup_x=x;

    Molecular_Dynamics(x,ham_init,ham_fin);
    metropolis = (double)rand()/RAND_MAX;
    if(exp(ham_init-ham_fin) > metropolis){
      //accept
      naccept=naccept+1;
    }else{
      //reject
      x=backup_x;

    }

    /****************/
    /*** data output ***/
    /****************/
    sum_xx=sum_xx+x*x;

    // output x, <x^2>, acceptance

    std::cout << x << ' ' << sum_xx/((double)(iter+1)) <<  ' ' <<
    ((double)naccept)/((double)iter+1) << std::endl;

    outputfile << x << ' ' << sum_xx/((double)(iter+1)) <<  ' ' <<
    ((double)naccept)/((double)iter+1) << std::endl;
```

```
    }

    outputfile.close();

    return 0;
}
```

At the beginning of the code, a few parameters are set. **niter** is the number of samples we will collect; **ntau** is N_τ; and **dtau** is $\Delta\tau$.

Then several routines/functions are defined:

- **BoxMuller** generates Gaussian random numbers by using the Box-Muller algorithm. We have to be careful about the normalization of the Gaussian here. It will be a kind of confusing when you go to complex variables; see the case of matrix integral in Sec. 4.2.3.

- **calc_action** calculates the action $S[x]$. In this case it is just $S[x] = \frac{x^2}{2}$. It is called in **calc_hamiltonian**.

- **calc_hamiltonian** adds $\frac{p^2}{2}$ to the action and returns the Hamiltonian. It is called in **Molecular_Dynamics**.

- **calc_delh** returns $\frac{dH}{dx} = \frac{dS}{dx} = x$. It is called in **Molecular_Dynamics**.

- **Molecular_Dynamics** performs one molecular evolution and returns the value of x after the evolution and the values of the Hamiltonian before and after the evolution.

When the action $S[x]$ is changed to more complicated functions, you have to rewrite **calc_action** and **calc_delh** accordingly.

In **main**, the only difference from Metropolis is that **Molecular_Dynamics** is used instead of a naive random change ($x \to x + \Delta x$ with random Δx), and the Metropolis test is performed by using ΔH instead of ΔS.

4.2.3 How it works, 2 — Matrix Integral

Next let us consider the same example as before,

$$S[\phi] = N\mathrm{Tr}\left(\frac{1}{2}\phi^2 + \frac{1}{4}\phi^4\right), \tag{20}$$

where ϕ is $N \times N$ Hermitian, and use the convention explained above. Then the force terms are

$$\frac{dP_{ij}}{d\tau} = -\frac{\partial S}{\partial \phi_{ji}} = -\phi_{ij} - \left(\phi^3\right)_{ij}, \qquad \frac{d\phi_{ij}}{d\tau} = P_{ij}. \tag{21}$$

The simulation code is very simple. Here is a one in Fortran 90:[13]

```fortran
program phi4

  implicit none
  !--------------------------------
  integer nmat
  parameter(nmat=100)
  integer ninit
  parameter(ninit=0)!ninit=1 -> new config; ninit=0 -> old config
  integer iter,niter
  parameter(niter=10000)
  integer ntau
  parameter(ntau=20)
  double precision dtau
  parameter(dtau=0.005d0)
  integer naccept
  double complex phi(1:NMAT,1:NMAT),backup_phi(1:NMAT,1:NMAT)
  double precision ham_init,ham_fin,action,sum_action
  double precision tr_phi,tr_phi2
  double precision metropolis

  open(unit=10,status='REPLACE',file='matrix-HMC.txt',action='WRITE')
  !************************************
  !*** Set the initial configuration ***
  !************************************
  call pre_random
  if(ninit.EQ.1)then
     phi=(0d0,0d0)
  else if(ninit.EQ.0)then
     open(UNIT=22, File ='config.dat', STATUS = "OLD", ACTION = "READ")
     read(22,*) phi
     close(22)
  end if
  sum_action=0d0
  !****************
  !*** Main part ***
  !****************
  naccept=0 !counter for the number of acceptance
  do iter=1,niter
```

---

[13]I realize that people grew up in the 21st century prefer C++. Still I personally love Fortran.

```fortran
      backup_phi=phi
      call Molecular_Dynamics(nmat,phi,dtau,ntau,ham_init,ham_fin)
      !**********************
      !*** Metropolis test ***
      !**********************
      call random_number(metropolis)
      if(dexp(ham_init-ham_fin) > metropolis)then
         !accept
         naccept=naccept+1
      else
         !reject
         phi=backup_phi
      end if
      !*******************
      !*** data output ***
      !*******************
      call calc_action(nmat,phi,action)
      sum_action=sum_action+action

      write(10,*)iter,action/dble(nmat*nmat),sum_action/dble(iter)/dble(nmat*nmat),&
       &dble(naccept)/dble(iter)

   end do

   close(10)

   open(UNIT = 22, File = 'config.dat', STATUS = "REPLACE", ACTION = "WRITE")
   write(22,*) phi
   close(22)

end program Phi4
```

Again, it is very similar to a Metropolis code; randomly change the configuration, perform the Metropolis test, randomly change the configuration, perform the Metropolis test,.... In **Molecular_Dynamics**, random momentum is generated with the normalization explained below (26), the molecular evolution performed, and H_i and H_f are calculated. Subroutines **calc_hamiltonian** and **calc_force** (which corresponds to **calc_delh** in the previous example) return the Hamiltonian and the force term $\frac{\partial H}{\partial \phi_{ji}} = \frac{\partial S}{\partial \phi_{ji}}$; it literally calculates products of matrices. Another subroutine **calc_action** is also simple. Let's see them one by one.[14]

---

[14]For routines which are not explained below, please look at the sample code at `https://github.com/MCSMC/MCMC_sample_codes`.

Molecular_Dynamics

```fortran
subroutine Molecular_Dynamics(nmat,phi,dtau,ntau,ham_init,ham_fin)

  implicit none

  integer nmat
  integer ntau
  double precision dtau
  double precision r1,r2
  double precision ham_init,ham_fin
  double complex phi(1:NMAT,1:NMAT)
  double complex P_phi(1:NMAT,1:NMAT)
  double complex delh(1:NMAT,1:NMAT)
  integer imat,jmat,step
  !*** randomly generate auxiliary momenta ***
  do imat=1,nmat-1
     do jmat=imat+1,nmat
        call BoxMuller(r1,r2)
        P_phi(imat,jmat)=dcmplx(r1/dsqrt(2d0))+dcmplx(r2/dsqrt(2d0))*(0D0,1D0)
        P_phi(jmat,imat)=dcmplx(r1/dsqrt(2d0))-dcmplx(r2/dsqrt(2d0))*(0D0,1D0)
     end do
  end do
  do imat=1,nmat
     call BoxMuller(r1,r2)
     P_phi(imat,imat)=dcmplx(r1)
  end do
  !*** calculate Hamiltonian ***
  call calc_hamiltonian(nmat,phi,P_phi,ham_init)
  !*** first step of leap frog ***
  phi=phi+P_phi*dcmplx(0.5d0*dtau)
  !*** 2nd, ..., Ntau-th steps ***
  step=1
  do while (step.LT.ntau)
     step=step+1
     call calc_force(delh,phi,nmat)
     P_phi=P_phi-delh*dtau
     phi=phi+P_phi*dcmplx(dtau)
  end do
  !*** last step of leap frog ***
  call calc_force(delh,phi,nmat)
```

```
    P_phi=P_phi-delh*dtau
    phi=phi+P_phi*dcmplx(0.5d0*dtau)
    !*** calculate Hamiltonian ***
    call calc_hamiltonian(nmat,phi,P_phi,ham_fin)

    return

END subroutine Molecular_Dynamics
```

The inputs are the matrix size **nmat**= N, the matrix **phi**= $\phi^{(k)}$, the step size and number of steps for the molecular evolution, **dtau**= $\Delta\tau$ and **ntau**= N_τ. The output is **phi**= ϕ' and **ham_init**= H_i, **ham_fin**= H_f. Note that the auxiliary momentum is neither input nor output; it is randomly generated every time in this subroutine.

Firstly random momentum P_ϕ is generated. **BoxMuller**(r_1,r_2) generates random numbers r_1 and r_2 with the Gaussian weight $\frac{e^{-r_1^2/2}}{\sqrt{2\pi}}$, $\frac{e^{-r_2^2/2}}{\sqrt{2\pi}}$. Note that P_ϕ is Hermitian, $P_\phi = P_\phi^\dagger$. Hence we take $P_{\phi,ii}$ to be real, $P_{\phi,ii} = r_1$, and $P_{\phi,ij} = P_{\phi,ji}^* = (r_1 + ir_2)/\sqrt{2}$ for $i < j$. A factor $1/\sqrt{2}$ is necessary in order to adjust the normalization.

```
    !*** randomly generate auxiliary momenta ***
    do imat=1,nmat-1
       do jmat=imat+1,nmat
          call BoxMuller(r1,r2)
          P_phi(imat,jmat)=dcmplx(r1/dsqrt(2d0))+dcmplx(r2/dsqrt(2d0))*(0D0,1D0)
          P_phi(jmat,imat)=dcmplx(r1/dsqrt(2d0))-dcmplx(r2/dsqrt(2d0))*(0D0,1D0)
       end do
    end do
    do imat=1,nmat
       call BoxMuller(r1,r2)
       P_phi(imat,imat)=dcmplx(r1)
    end do
```

Then we calculate the initial value of the Hamiltonian:

```
    !*** calculate Hamiltonian ***
    call calc_hamiltonian(nmat,phi,P_phi,ham_init)
```

Because we have already taken a backup of ϕ before using this subroutine, we do not take a backup here.

Then we perform the molecular evolution by using the leap frog method.

```
    !*** first step of leap frog ***
    phi=phi+P_phi*dcmplx(0.5d0*dtau)
```

33

Note that we need a factor 1/2 here! Then we just repeat the leap-frog steps,

```
!*** 2nd, ..., Ntau-th steps ***
step=1
do while (step.LT.ntau)
   step=step+1
   call calc_force(delh,phi,nmat)
   P_phi=P_phi-delh*dtau
   phi=phi+P_phi*dcmplx(dtau)
end do
```

and we need a factor 1/2 again at the end:

```
!*** last step of leap frog ***
call calc_force(delh,phi,nmat)
P_phi=P_phi-delh*dtau
phi=phi+P_phi*dcmplx(0.5d0*dtau)
```

Now the molecular evolution has been done. In order to perform the Metropolis test, we need to calculate H_f:

```
!*** calculate Hamiltonian ***
call calc_hamiltonian(nmat,phi,P_phi,ham_fin)
```

Next we need to understand other subroutines called in this subroutine. We will skip **BoxMuller** because it is exactly the same as before. The other three will be explained below; they are almost trivial as well though.

calc_force

This subroutine calculates the force term

$$\frac{\partial H}{\partial \phi_{ji}} = \frac{\partial S}{\partial \phi_{ji}} = N \left(\phi + \phi^3 \right)_{ij}. \tag{22}$$

We just do it without using thinking too much, in the following manner:

```
subroutine calc_force(delh,phi,nmat)

  implicit none

  integer nmat
  double complex phi(1:NMAT,1:NMAT),phi2(1:NMAT,1:NMAT),phi3(1:NMAT,1:NMAT)
```

```fortran
   double complex delh(1:NMAT,1:NMAT)

   integer imat,jmat,kmat
   !*** phi2=phi*phi, phi3=phi*phi*phi ***
   phi2=(0d0,0d0)
   phi3=(0d0,0d0)
   do imat=1,nmat
      do jmat=1,nmat
         do kmat=1,nmat
            phi2(imat,jmat)=phi2(imat,jmat)+phi(imat,kmat)*phi(kmat,jmat)
         end do
      end do
   end do
   do imat=1,nmat
      do jmat=1,nmat
         do kmat=1,nmat
            phi3(imat,jmat)=phi3(imat,jmat)+phi2(imat,kmat)*phi(kmat,jmat)
         end do
      end do
   end do
   !*** delh=dH/dphi ***
   delh=phi+phi3
   delh=delh*dcmplx(nmat)

   return

END subroutine Calc_Force
```

calc_hamiltonian

This subroutine just returns $H = \frac{1}{2}\mathrm{Tr}P^2 + S[\phi]$. Firstly another subroutine **calc_action**, which calculates the action, is called. Then $\frac{1}{2}\mathrm{Tr}P^2$ is added. It is too simple and looks almost stupid, but most things needed for MCMC codes are like this.

```fortran
SUBROUTINE calc_hamiltonian(nmat,phi,P_phi,ham)

   implicit none

   integer nmat
   double precision action,ham
   double complex phi(1:NMAT,1:NMAT)
   double complex P_phi(1:NMAT,1:NMAT)
```

```
   integer imat,jmat

   call calc_action(nmat,phi,action)

   ham=action
   do imat=1,nmat
      do jmat=1,nmat
         ham=ham+0.5d0*dble(P_phi(imat,jmat)*P_phi(jmat,imat))
      end do
   end do

   return

END SUBROUTINE calc_hamiltonian
```

calc_action

This subroutine just returns $S[\phi]$. We honestly write down everything. It is tedious but straightforward. You just have to be patient.

```
SUBROUTINE calc_action(nmat,phi,action)

   implicit none

   integer nmat
   double precision action
   double complex phi(1:NMAT,1:NMAT)
   double complex phi2(1:NMAT,1:NMAT)
   integer imat,jmat,kmat
   !*** phi2=phi*phi ***
   phi2=(0d0,0d0)
   do imat=1,nmat
      do jmat=1,nmat
         do kmat=1,nmat
            phi2(imat,jmat)=phi2(imat,jmat)+phi(imat,kmat)*phi(kmat,jmat)
         end do
      end do
   end do

   action=0d0
   !*** Tr phi^2 term ***
   do imat=1,nmat
      action=action+0.5d0*dble(phi2(imat,imat))
```

N_τ	acceptance	acceptance/N_τ
4	0.0633	0.01583
6	0.3418	0.05697
8	0.6023	0.07529
10	0.7393	0.07393
20	0.9333	0.04667

Table 2: Acceptance rate for several choices of N_τ, with $N_\tau \Delta\tau = 0.1$, matrix size $N = 100$. We started the measurement runs with well thermalized configurations and collected 10000 samples for each parameter choice.

```
  end do
  !*** Tr phi^4 term ***
  do imat=1,nmat
     do jmat=1,nmat
        action=action+0.25d0*dble(phi2(imat,jmat)*phi2(jmat,imat))
     end do
  end do
  !*** overall normalization ***
  action=action*dble(nmat)

  return

END SUBROUTINE calc_action
```

Simulation

In order to see how we can adjust the simulation parameters, let us vary N_τ and $\Delta\tau$ keeping the product $N_\tau \Delta\tau$ to be 0.1. Then the acceptance rate changes as follows shown in Table 2. Roughly speaking, the simulation cost is proportional to N_τ. When $N_\tau \Delta\tau$ is fixed, the matrix ϕ changes more or less the same amount by the molecular evolution, regardless of N_τ. Therefore, the rate of change is proportional to the acceptance rate. Hence the change per cost is (acceptance)/N_τ. We should maximize it. So we should use $N_\tau = 8$ or 10. In Fig. 14 we have plotted $\langle S/N^2 \rangle = \frac{1}{n} \sum_{k=1}^{n} S[\phi^{(k)}]$ for several different values of N_τ by taking the horizontal axis to be 'cost'= $n \times N_\tau$. We can see that $N_\tau = 8, 10$ are actually cost effective.

Ideally we should do similar cost analysis varying $N_\tau \Delta\tau$, and estimate the autocorrelation length as well, to achieve *more independent configurations with less cost*.

Before closing this section, let us demonstrate the importance of the leap-frog method.
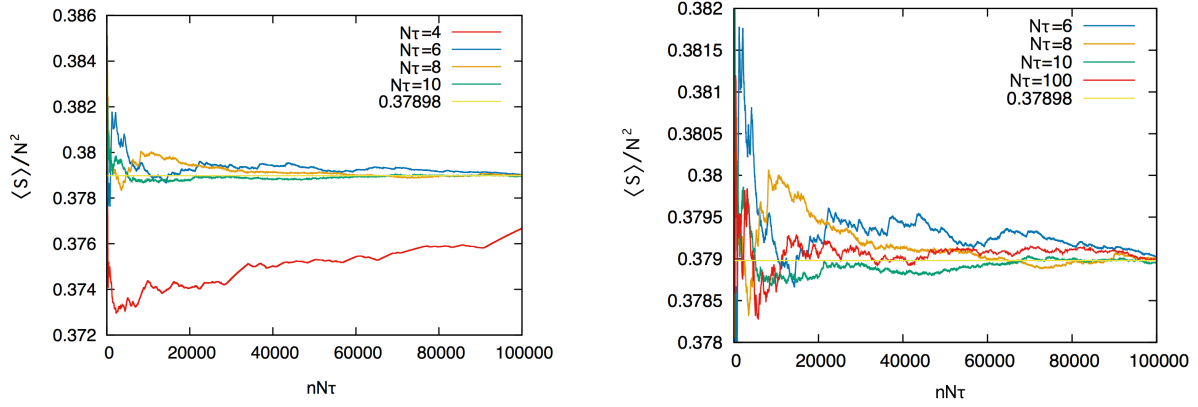
Figure 14: $\langle S \rangle / N^2 = \frac{1}{n} \sum_{k=1}^{n} S[\phi^{(k)}]$ with $N = 100$, for several different values of N_τ. The horizontal axis is $n \times N_\tau$, which is proportional to the cost (time and electricity needed for the simulation). We can see that $N_\tau = 8, 10$ are more cost effective; i.e. better convergence with less cost. Note that we started the measurement runs with well thermalized configurations.

Let us try a wrong algorithm: we omit a factor $1/2$ in the final step in Fig. 13.[15] The outcome is a disaster; as shown in Fig. 15, different N_τ and $\Delta\tau$ give different values. Correct expectation value (which agree with the value in Fig. 14 up to a very small $1/N$ correction) is obtained only at $N_\tau = \infty$ with $N_\tau \Delta\tau$ fixed. It is also easy to see the importance of the normalization of the auxiliary momentum; you can try it by yourself.

Remarks on the normalization

Above we have assumed that the variables $x_1, x_2, \cdots$ are real. When you have to deal with complex variables, Hermitian matrices etc, you can always rewrite everything by using real variables; for example one can write a Hermitian matrix M as $M = \sum_a M_a T^a$, where T^a are generators and M_a are real-valued coefficients. But it is tedious and we have seen so many lattice QCD practitioners, who almost always work on SU(3), waste time struggling with SU(N), just to fix the normalization. So let us summarize the cautions regarding the normalization.

Let us consider the simplest case again: the Gaussian integral, $S[x] = \frac{x^2}{2}$. The Hamiltonian is $H[x, p] = \frac{x^2}{2} + \frac{p^2}{2}$, and the equations of motion are

$$\frac{dp}{d\tau} = -\frac{\partial H}{\partial x} = -\frac{\partial S}{\partial x} = -x, \qquad \frac{dx}{d\tau} = \frac{\partial H}{\partial p} = p. \tag{23}$$

This p should be generated with the weight $\frac{1}{\sqrt{2\pi}} e^{-p^2/2}$.

---

[15]This was the bug in my first HMC code. It took several days to find it.
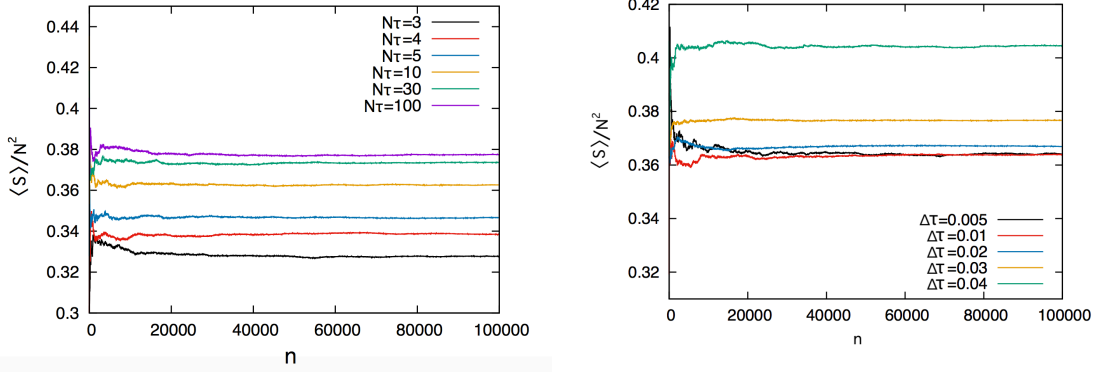
Figure 15: $\langle S \rangle / N^2 = \frac{1}{n} \sum_{k=1}^{n} S[\phi^{(k)}]$ with $N = 10$, for several different values of N_τ and $\Delta\tau$, *without a factor $1/2$ in the final step of the leap frog.* (Left) $N_\tau \Delta\tau$ is fixed to 0.1; (Right) N_τ is fixed to 10. Correct expectation value is obtained only at $N_\tau = \infty$ with $N_\tau \Delta\tau$ fixed.

Now let x be complex and $S[x] = |x|^2 = \bar{x}x$. Let p be the conjugate of $\bar{x}$, then the Hamiltonian is $H[x, p] = \bar{x}x + \bar{p}p$, and

$$\frac{dp}{d\tau} = -\frac{\partial H}{\partial \bar{x}} = -\frac{\partial S}{\partial \bar{x}} = -x, \qquad \frac{dx}{d\tau} = \frac{\partial H}{\partial \bar{p}} = p. \tag{24}$$

To rewrite it to real variables with the right normalization, we do as follows:

$$x = \frac{x_R + ix_I}{\sqrt{2}}, \qquad p = \frac{p_R + ip_I}{\sqrt{2}}. \tag{25}$$

Then $H[x, p] = \frac{x_R^2 + x_I^2 + p_R^2 + p_I^2}{2}$, and (x_R, p_R) and (x_I, p_I) are conjugate pairs. We should generate p_R and p_I with weight $\frac{1}{\sqrt{2\pi}} e^{-(p_R)^2/2}$ and $\frac{1}{\sqrt{2\pi}} e^{-(p_I)^2/2}$.

Now let X_{ij} be a Hermitian matrix, $X_{ji} = X_{ij}^*$. The conjugate P is also a Hermitian matrix, and we can take $P_{ji} = P_{ij}^*$ to be the conjugate of X_{ij}. A simple Hamiltonian $H = \frac{1}{2}\text{Tr}X^2 + \frac{1}{2}\text{Tr}P^2$ becomes

$$H = \frac{1}{2}\sum_i \left(X_{ii}^2 + P_{ii}^2\right) + \sum_{i<j} \left(X_{ij}X_{ij}^* + P_{ij}P_{ij}^*\right). \tag{26}$$

Hence P_{ii} should be generated with the weight $\frac{1}{\sqrt{2\pi}} e^{-(P_{ii})^2/2}$, while P_{ij} can be obtained by rewriting it as $P_{ij} = \frac{P_{ij,R} + iP_{ij,I}}{\sqrt{2}}$ and generating $P_{ij,R}$, $P_{ij,I}$ with the weight $\frac{1}{\sqrt{2\pi}} e^{-(P_{ij,R})^2/2}$, $\frac{1}{\sqrt{2\pi}} e^{-(P_{ij,I})^2/2}$. The force terms are as follows:

$$\frac{dP_{ij}}{d\tau} = -\frac{\partial H}{\partial X_{ji}} = -\frac{\partial S}{\partial X_{ji}} = -X_{ij}, \qquad \frac{dX_{ij}}{d\tau} = \frac{\partial H}{\partial P_{ji}} = P_{ij}. \tag{27}$$

39

As the final example, let X_{ij} be a complex matrix. The conjugate P is also a Hermitian matrix, and we can take P_{ij}^* to be the conjugate of X_{ij}. A simple Hamiltonian $H = \text{Tr} X^\dagger X + \text{Tr} P^\dagger P$ becomes

$$H = \sum_{i,j} \left(X_{ij} X_{ij}^* + P_{ij} P_{ij}^* \right). \tag{28}$$

Hence P_{ij} can be obtained by rewriting it as $P_{ij} = \frac{P_{ij,R} + i P_{ij,I}}{\sqrt{2}}$ and generating $P_{ij,R}$, $P_{ij,I}$ with the weight $\frac{1}{\sqrt{2\pi}} e^{-(P_{ij,R})^2/2}$, $\frac{1}{\sqrt{2\pi}} e^{-(P_{ij,I})^2/2}$. The force terms are as follows:

$$\frac{dP_{ij}}{d\tau} = -\frac{\partial S}{\partial X_{ij}^*} = -X_{ij}, \qquad \frac{dX_{ij}}{d\tau} = \frac{\partial S}{\partial P_{ij}^*} = P_{ij}. \tag{29}$$

The case with generic actions S should be apparent. Note that, when you change the normalization of the p^2 term in the Hamiltonian, you have to change the width of random Gaussian appropriately. Otherwise you will end up in getting wrong answers.

Remark on debugging

One extra bonus associated with HMC is that we can use the conservation of the Hamiltonian for debugging. It is very rare to make bugs in the calculations of the action and force term consistently; so, practically, unless we code both correctly we cannot see the conservation of the Hamiltonian in the 'continuum limit' $N_\tau \to \infty$, $\Delta\tau \to 0$ with $N_\tau \Delta\tau$ fixed.[16] This is a very good check of the code; see Fig. 16.

4.3 Multiple step sizes

Let us conside a two-matrix model

$$S[\phi_1, \phi_2] = N\text{Tr}\left(V_1(\phi_1) + V_2(\phi_2) + \phi_1 \phi_2\right), \tag{30}$$

where

$$V_1(\phi_1) = \frac{m_1^2}{2}\phi_1^2 + \frac{1}{4}\phi_1^4, \qquad V_2(\phi_2) = \frac{m_2^2}{2}\phi_2^2 + \frac{1}{4}\phi_2^4. \tag{31}$$

Imagine an extreme situation, $m_1 = 1$ and $m_2 = 1000000000$, i.e. ϕ_2 is much heavier. Then typical value of ϕ_1 is much larger than that of ϕ_2. If we use the same step size for both, then in order to raise the acceptance rate for ϕ_2 we have to take the step size very small, which leads to a very long autocorrelation for ϕ_1.

As we have emphasized in Sec. 3.4.2, the choice of the step size has to be consistent with the requirements listed in Sec. 2, but other than that it is completely arbitrary; see

---

[16]Note that the normalization of the auxiliary momentum and the factor $1/2$ at the first and last steps of the leap-frog evolution cannot be tested by the conservation of the Hamiltonian.
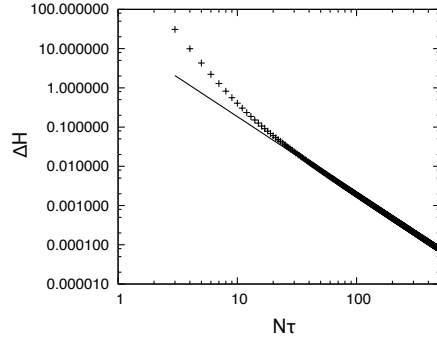
Figure 16: $\Delta H = H_f - H_i$ vs N_τ, with $N_\tau \Delta \tau = 0.1$ (fixed), in log-log scale. We picked up a thermalized configuration $\{\phi\}$ and a randomly generated auxiliary momentum $\{P_\phi\}$, and used the same $\{\phi, P_\phi\}$ for all $(N_\tau, \Delta \tau)$. The solid line is $18.5 N_\tau^{-2}$. The action is (20) with $N = 100$. When you confirm the conservation of the Hamiltonian, debugging is more or less done.

Sec. 4.1. Hence we can simply use different step sizes for ϕ_1 and ϕ_2. This simple fact is very important in the simulation of QFT: we should use larger step size for lighter particle. Also, in the momentum space, high-frequency modes are 'heavier' in that $m^2 + p^2$ behaves like a mass. Hence we should use smaller step size for ultraviolet modes, larger step size for infrared modes. This method is called 'Fourier acceleration'.

4.4   Different algorithms for different fields

We can even use different update algorithms for different variables, and as we will see, this is important when we study systems with fermions.

Let us consider

$$S(x,y) = y^2 f(x) + g(x) \tag{32}$$

where $f(x)$ and $g(x)$ are complicated functions. Then we can repeat the following two steps,

- Update y for fixed x,

- Update x for fixed y.

In Sec. 4.1 we introduced essentially the same example, namely we adopted the Metropolis algorithm and varied $x_1, x_2, \cdots$ one by one.

If $f(x) > 0$, then $z \equiv y\sqrt{f(x)}$ has a Gaussian weight for each fixed x. In this case, we can use the Box-Muller algorithm (Appendix B) to generate z; then there is no autocorrelation between y's. Hence we can use the following method:

- Update y for fixed x, by generating Gaussian random z and setting $y = z/\sqrt{f(x)}$.

- Update x for fixed y, by using Metropolis or HMC.

Note that the use of the Box-Muller algorithm does not violate the conditions listed in Sec. 2. It just gives us a very special Markov chain without autocorrelation.

4.5   QFT example 1: 4d scalar theory

Let us consider 4d scalar theory,

$$S[\phi] = N \int d^4x \mathrm{Tr} \left(\frac{1}{2} \left(\partial_\mu \phi \right)^2 + \frac{m^2}{2} \phi^2 + V(\phi) \right), \tag{33}$$

where the $N \times N$ Hermitian matrices $\phi(x)$ now depends on the coordinate x. For simplicity we assume the spacetime is compactified to a square four-torus with circumference ℓ and

volume $V = \ell^4$. It can be regularized by using an n^4 lattice with the lattice spacing $a = L/n$ as

$$S_{\text{lattice}}[\phi] = Na^4 \sum_{\vec{x}} \text{Tr} \left(\frac{1}{2} \sum_{\mu} \left(\frac{\phi_{\vec{x}+\hat{\mu}} - \phi_{\vec{x}}}{a} \right)^2 + \frac{m^2}{2} \phi_{\vec{x}}^2 + V(\phi_{\vec{x}}) \right), \tag{34}$$

where $\hat{\mu}$ stands for a shift of one lattice unit along the μ direction. The path-integral is just a multi-variable integral with Hermitian matrices, the methods we have already explained can directly be applied.

4.6 QFT example 2: Wilson's plaquette action (SU(N) pure Yang-Mills)

Let's move on to 4d SU(N) Yang-Mills. The continuum action we consider is

$$S_{\text{continuum}} = \frac{1}{4g_{YM}^2} \int d^4x \text{Tr} F_{\mu\nu}^2, \tag{35}$$

where the field strength $F_{\mu\nu}$ is defined by $F_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu + i[A_\mu, A_\nu]$. Typically we take the 't Hooft coupling $\lambda = g_{YM}^2 N$ fixed when we take large N. As a lattice regularization we use Wilson's plaquette action,[17]

$$S_{\text{lattice}} = -\beta N \sum_{\mu \neq \nu} \sum_{\vec{x}} \text{Tr} \, U_{\mu,\vec{x}} U_{\nu,\vec{x}+\hat{\mu}} U_{\mu,\vec{x}+\hat{\nu}}^\dagger U_{\nu,\vec{x}}^\dagger. \tag{36}$$

Here $U_{\mu,\vec{x}}$ is a unitary variable living on a link connecting to lattice points $\vec{x}$ and $\vec{x} + a\hat{\mu}$, where a is the lattice spacing and $\hat{\mu}$ is a unit vector along the μ-th direction. It is related to the gauge field $A_\mu(\vec{x})$ by $U_{\mu,\vec{x}} = e^{iaA_\mu(\vec{x})}$. The lattice coupling constant β is the inverse of the 't Hooft coupling, $\beta = 1/\lambda$, and it should be scaled appropriately with the lattice spacing a in order to achieve the right continuum limit.

4.6.1 Metropolis for unitary variables

The only difference is that, instead of adding random numbers, we should multiply random unitary matrices. A random unitary matrix can be generated as follows. Firstly, we generate random Hermitian matrix H by using random numbers. For example we can generate it with Gaussian weight $\sim e^{-\text{Tr}H^2/2\sigma^2}$. Then $V = e^{iH}$ is random unitary centered around $V = 1$. When σ is small, it is more likely to be close to 1. Hence σ is 'step size'. Of course, you can use the uniform random number to generate H if you want. Regardless, the Metropolis goes as follows:

---

[17]Often the overall factor N is included in β and $\beta' = \beta N$ is used as the lattice coupling. It is simply a bad convention when we consider generic values of N, because the coupling to be fixed as N is varied is not β' but β.

1. Generate V randomly, change $U_{1,\vec{x}}$ to $U'_{1,\vec{x}} = U_{1,\vec{x}}V$, and perform the Metropolis test.

2. Do the same for $U_{2,\vec{x}}$, $U_{3,\vec{x}}$ and $U_{4,\vec{x}}$.

3. Do the same for other lattice sites.

4. Repeat the same procedure many many times.

4.6.2 HMC for Wilson's plaquette action

HMC for unitary variables goes as follows. Let us define the momentum p_μ^{ij} conjugate to the gauge field A_μ^{ji} by

$$\frac{dU}{d\tau} = ipU, \qquad \frac{dU^\dagger}{d\tau} = -iUp. \tag{37}$$

It generates

$$U \to e^{i\delta A}U, \qquad U^\dagger \to U^\dagger e^{-i\delta A}. \tag{38}$$

Therefore,

$$\frac{dp_{ij}}{d\tau} = -\frac{\partial S}{dA^{ji}} = -i\left(U\frac{\partial S}{\partial U}\right)_{ij} + i\left(U\frac{\partial S}{\partial U}\right)^*_{ji} \tag{39}$$

where the second term comes from the derivative w.r.t. $U^\dagger$. The discrete molecular evolution can be defined as follows:

1.

$$U(\Delta\tau/2) = \exp\left(i\frac{\Delta\tau}{2}p(0)\right) \cdot U(0),$$

$$p(\Delta\tau) = p(0) + \Delta\tau \cdot \frac{dp}{d\tau}(\Delta\tau/2).$$

2. Repeat the following for $\tau = \Delta\tau, 2\Delta\tau, \cdots, (N_\tau - 1)\Delta\tau$:

$$U(\tau + \Delta\tau/2) = \exp\left(i\Delta\tau p(\tau)\right) \cdot U(\tau - \Delta\tau/2),$$

$$p(\tau + \Delta\tau) = p(\tau) + \Delta\tau \cdot \frac{dp}{d\tau}(\tau + \Delta\tau/2).$$

3.

$$U(N_\tau\Delta\tau) = \exp\left(i\frac{\Delta\tau}{2}p(N_\tau\Delta\tau)\right) \cdot U\left((N_\tau - 1/2)\Delta\tau\right).$$

In order to calculate $e^{i\Delta\tau p}$, it is necessary to diagonalize p. As long as one considers $SU(N)$ theory with not very large N, the diagonalization is not that costly. (Note also that, when the fermions are introduced, this part cannot be a bottle-neck, so we do not have to care; we should spend our effort for improving other parts.) In case we need to cut the cost as much as possible, we can approximate it by truncating the Taylor expansion of $e^{i\Delta\tau \cdot p}$ at some finite order.

5 Including fermions with HMC and RHMC

Let us consider the simplest example,

$$S[x, \psi, \bar{\psi}] = \frac{x^2}{2} + \bar{\psi} D(x) \psi, \tag{40}$$

where ψ is a complex Grassmann number and a function $D(x)$ is a 'Dirac operator'. We can integrate out ψ by hand, so that

$$Z = \int dx d\psi d\bar{\psi} e^{-S[x,\psi,\bar{\psi}]} = \int dx D(x) e^{-x^2/2} = \int dx e^{-x^2/2 + \log D(x)}. \tag{41}$$

Hence we need to deal with the effective action in terms of x, $S_{\mathrm{eff}} = \frac{x^2}{2} - \log D(x)$. This is simple enough so that we don't need anything more sophisticated than the Metropolis algorithm. However our life becomes a bit more complicated when there are multiple variables, $x_{1,2,\cdots,n_x}$ and $\psi_{1,2,\cdots,n_\psi}$. Let us consider the action of the following form,

$$S[x, \psi, \bar{\psi}] = \sum_{i=1}^{n_x} \frac{x_i^2}{2} + \sum_{a,b=1}^{n_\psi} \bar{\psi}_a D_{ab}(x) \psi_b. \tag{42}$$

Now the Dirac operator $D_{ab}(x)$ is an $n_\psi \times n_\psi$ matrix, and the partition function becomes

$$Z = \int [dx][d\psi][d\bar{\psi}] e^{-S[x,\psi,\bar{\psi}]} = \int [dx] \det D(x) \cdot e^{-\sum_i x_i^2/2}. \tag{43}$$

We can still use the Metropolis algorithm in principle,[18] but the calculation of the determinant is very costly (cost$\sim n_\psi^3$). Even worse, though $D_{ab}(x)$ is typically sparse, it is not easy to utilize the sparseness when one calculates the determinant. HMC and RHMC avoid the evaluation of the determinant and enable us to utilize the sparseness of $D_{ab}(x)$.

5.1 2-flavor QCD with HMC

Let us consider 2-flavor QCD, whose action in the continuum is

$$S[A_\mu, \psi_f, \bar{\psi}_f] = \int d^4 x \left(\frac{1}{4} Tr F_{\mu\nu}^2 + \sum_{f=1}^{2} \bar{\psi}^{(f)} D^{(f)} \psi^{(f)} \right), \tag{44}$$

where $A_\mu (\mu = 1, 2, 3, 4)$ is the SU(3) gauge field and $D_f = \gamma^\mu D_\mu + m_f$ is the Dirac operator with fermion mass m_f. For a lattice regularization, we use link variables as in Sec.4.6, and put fermions on sites. The latice action we consider is

$$S_{\mathrm{Lattice}}[U_\mu, \psi_f, \bar{\psi}_f] = S_B[U_\mu] + \sum_{\vec{x}, \vec{y}} \sum_{f=1}^{2} \bar{\psi}_{\vec{x}\alpha}^{(f)} D_{\vec{x}\alpha, \vec{y}\beta}^{(f)} \psi_{\vec{y}\beta}^{(f)}, \tag{45}$$

---

[18]Here we assumed $\det D(x) > 0$ for any x, i.e. the sign problem is absent.

where $S_B[U_\mu]$ is the plaquette action (36). There are various choices for a lattice Dirac operator $D^{(f)}_{\vec{x}\alpha,\vec{y}\beta}$; we do not specify it here.

Below we assume $m_1 = m_2 = m$ (i.e. neglect the difference of the mass of up and down quarks), then the partition function is

$$Z_{\text{Lattice}} = \int [dU] \, (\det D)^2 \, e^{-S_B}. \tag{46}$$

Usually the determinant is real, and hence $(\det D)^2 = \det(D^\dagger D)$.

Now we introduce *pseudo-fermion* F, on which the Dirac operator acts just in the same way as on ψ. But F is a complex bosonic field. Then

$$Z_{\text{Lattice}} = \int [dU][dF] e^{-S_B - F^\dagger (D^\dagger D)^{-1} F}. \tag{47}$$

At first sight it may look like a stupid way of writing an easy thing in a complicated form. However it enables us to avoid the calculation of $\det D$; rather we have to calculate a linear equation

$$(D^\dagger D)\chi = F, \tag{48}$$

for which the sparseness of D can be fully utilized. In order to obtain the solution χ, we can use the conjugate gradient method; see Appendix A.

The strategy is simply applying the HMC algorithm to

$$S[U_\mu, F] = S_B[U_\mu] + F^\dagger (D^\dagger D)^{-1} F, \tag{49}$$

by using A_μ and F as dynamical variables. The only nontrivial parts are the calculation of the force and Hamiltonian. For that, we only need the solution χ of $(D^\dagger D)\chi = F$. Indeed, the force is calculated as

$$-\frac{\partial S}{\partial A_\mu} = -\frac{\partial S_B}{\partial A_\mu} + \chi^\dagger \frac{\partial (D^\dagger D)}{\partial A_\mu} \chi \tag{50}$$

and

$$-\frac{\partial S}{\partial F^\dagger} = \chi, \tag{51}$$

and the action is simply

$$S[U_\mu, F] = S_B[U_\mu] + F^\dagger \chi. \tag{52}$$

A better way of treating F

It is possible to update the pseudo-fermion F more efficiently, by using the idea explained in Sec.4.4. This is based on a simple observation that $\Phi \equiv (D^\dagger)^{-1} F$ has the Gaussian weight and hence can easily be generated randomly by using the Gaussian random number generator.

Hence we can do as follows:

1. Randomly generate $\Phi^{(k+1)}$ with the Gaussian weight[19] $e^{-\Phi^\dagger \Phi}$.

2. Calculate the pseudo-fermion $F^{(k+1)} = D^\dagger(U_\mu^{(k)}) \cdot \Phi^{(k+1)}$.

3. Randomly generate auxiliary momenta $P_\mu^{(k)}$, which are conjugate to $A_\mu^{(k)}$, with probabilities proportional to $e^{-\mathrm{Tr}(P_\mu^{(k)})^2/2}$.

4. Calculate

$$H_i = S_B[U^{(k)}] + \sum_\mu \mathrm{Tr}(P_\mu^{(k)})^2/2 + (F^{(k+1)})^\dagger (D^\dagger(U^{(k)}) \cdot D(U^{(k)}))^{-1} F^{(k+1)}. \tag{53}$$

Note that $(F^{(k+1)})^\dagger (D^\dagger(U^{(k)}) \cdot D(U^{(k)}))^{-1} F^{(k+1)} = (\Phi^{(k+1)})^\dagger \Phi^{(k+1)}$.

5. Then perform the Molecular evolution for U_μ, fixing F to be $F^{(k+1)}$. The force is

$$-\frac{\partial S_B}{\partial A_\mu} + \chi^\dagger \frac{\partial (D^\dagger D)}{\partial A_\mu} \chi, \tag{54}$$

where $(D^\dagger D)\chi = F$ as before. Then we obtain $U_\mu^{(k+1)}$ and $P_\mu^{(k+1)}$.

6. Calculate

$$H_f = S_B[U^{(k+1)}] + \sum_\mu \mathrm{Tr}(P_\mu^{(k+1)})^2/2 + (F^{(k+1)})^\dagger (D^\dagger(U^{(k+1)}) \cdot D(U^{(k+1)}))^{-1} F^{(k+1)}. \tag{55}$$

7. Metropolis test: Generate a uniform random number r between 0 and 1. If $r < e^{H_i - H_f}$, $U_\mu^{(k+1)} = U_\mu^{(k)}(\tau_f)$, otherwise $U_\mu^{(k+1)} = U_\mu^{(k)}$.

8. Repeat 1–7.

5.2 (2+1)-flavor QCD with RHMC

Strange quark has much bigger mass than up and down quarks. So let us take $m_1 = m_2 \ll m_3 = m_s$. Let us denote the Dirac operators for up and down by D, and the one for strange by D_s. The partition function is

$$Z_{\mathrm{Lattice}} = \int [dU] \, (\det D_s) \, (\det D)^2 \, e^{-S_B}. \tag{56}$$

---

[19]By using the Box-Muller algorithm explained in Appendix B, real Gaussian random numbers with variance 1 is generated, i.e. the weight of $x, y \in \mathbb{R}$ is $e^{-x^2/2}$ and $e^{-y^2/2}$. By dividing $\tilde{x} = x/\sqrt{2}$ and $\tilde{y} = y/\sqrt{2}$ have the weights $e^{-\tilde{x}^2}$ and $e^{-\tilde{y}^2}$. To reproduce the weight is $e^{-\Phi^\dagger \Phi} = e^{-(\mathrm{Re}\Phi)^2 - (\mathrm{Im}\Phi)^2}$, we should take $\Phi = (\tilde{x} + \sqrt{-1}\tilde{y}) = (x + \sqrt{-1}y)/\sqrt{2}$. If you use a wrong normalization, you end up in a wrong result.

Hence we need to introduce two pseudo fermions F and F_s as[20]

$$S[U_\mu, F] = S_B[U_\mu] + F^\dagger (D^\dagger D)^{-1} F + F_s^\dagger (D_s^\dagger D_s)^{-1/2} F_s. \tag{57}$$

The last term in the right hand side is problematic because it is difficult (or costly) to solve $(D^\dagger D)^{1/2} \chi_s = F_s$.

The Rational Hybrid Monte Carlo (RHMC) algorithm [15] evades this problem by utilizing a rational approximation,

$$(D^\dagger D)^{-1/2} \simeq a_0 + \sum_{i=1}^{Q} \frac{a_i}{D^\dagger D + b_i}, \tag{58}$$

where a's and b's are positive constants, and Q is typically 10 or 20. These numbers are adjusted so that the rational approximation is good for all samples appearing in the actual simulation. A good code to find such numbers can be found at [17].

By using this rational approximation, we replace the last term by

$$S_{\text{strange}} = a_0 F_s^\dagger F_s + \sum_{i=1}^{Q} a_i F_s^\dagger \frac{1}{D_s^\dagger D_s + b_i} F_s. \tag{59}$$

To determine the force, we need to solve $(D_s^\dagger D_s + \beta_i) \chi_{s,i} = F_s$. Rather surprisingly, we do not have to solve these equations Q times; the multi-mass CG solver [16] solves all Q equations simultaneously, essentially without any additional cost. See Appendix A.

1. Randomly generate $\Phi^{(k+1)}$ and $\Phi_s^{(k+1)}$ with the Gaussian weights $e^{-\Phi^\dagger \Phi}$ and $e^{-\Phi_s^\dagger \Phi_s}$.

2. Calculate the pseudo-fermion $F^{(k+1)} = D^\dagger (U_\mu^{(k)}) \cdot \Phi^{(k+1)}$.

3. Calculate the pseudo-fermion $F_s^{(k+1)} = \left(D^\dagger (U_\mu^{(k)}) \cdot D(U_\mu^{(k)}) \right)^{1/4} \Phi_s^{(k+1)}$. See the end of this section for an efficient way to do it.

4. Randomly generate auxiliary momenta $P_\mu^{(k)}$, which are conjugate to $A_\mu^{(k)}$, with probabilities proportional to $e^{-\text{Tr}(P_\mu^{(k)})^2/2}$.

5. Calculate

$$H_i = H[U^{(k)}, P^{(k)}, F^{(k+1)}, F_s^{(k+1)}] \tag{60}$$

where

$$
\begin{aligned}
H[U, P, F, F_s] &= S_B[U] + \sum_\mu \text{Tr}(P_\mu)^2/2 \\
&\quad + F^\dagger (D^\dagger(U) \cdot D(U))^{-1} F
\end{aligned}
$$

---

[20]$D_s^\dagger D_s$ is used so that the action becomes positive definite.

$$+ a_0 F_s^\dagger F_s + \sum_{i=1}^{Q} a_i F_s^\dagger \frac{1}{D_s(U)^\dagger D_s(U) + b_i} F_s. \tag{61}$$

Note that, although the last line formally seems to agree with $(\Phi_s^{(k+1)})^\dagger \Phi_s^{(k+1)}$ at $\tau = 0$, it is not the case due to the approximations used for $(D^\dagger D)^{-1/2}$ and $(D^\dagger D)^{+1/4}$ are not exactly the 'inverse'.

6. Then perform the Molecular evolution for U_μ, fixing F to be $F^{(k+1)}$. The force is

$$-\frac{\partial S_B}{\partial A_\mu} + \chi^\dagger \frac{\partial (D^\dagger D)}{\partial A_\mu} \chi + \sum_{i=1}^{Q} a_i \chi_{s,i}^\dagger \frac{\partial (D_s^\dagger D_s)}{\partial A_\mu} \chi_{s,i}, \tag{62}$$

where $(D^\dagger D)\chi = F$ and $(D_s^\dagger D_s + b_i)\chi_{s,i} = F_s$. Then we obtain $U_\mu^{(k+1)}$ and $P_\mu^{(k+1)}$.

7. Calculate

$$H_f = H[U^{(k+1)}, P^{(k+1)}, F^{(k+1)}, F_s^{(k+1)}] \tag{63}$$

8. Metropolis test: Generate a uniform random number r between 0 and 1. If $r < e^{H_i - H_f}$, $U_\mu^{(k+1)} = U_\mu^{(k)}(\tau_f)$, i.e. the new configuration is 'accepted.' Otherwise $U_\mu^{(k+1)} = U_\mu^{(k)}$, i.e. the new configuration is 'rejected.'

9. Repeat 1 – 8.

How to calculate $F = \left(D^\dagger D\right)^{1/4} \Phi$

We can use the rational approximation

$$x^{1/4} \simeq a_0' + \sum_{k=1}^{Q'} \frac{a_k'}{x + b_k'}. \tag{64}$$

Then

$$F \simeq a_0' \Phi + a_k' \chi_k', \tag{65}$$

where

$$\frac{1}{D_s^\dagger D_s + b_k'} \Phi = \chi_k' \tag{66}$$

is obtained by solving

$$(D_s^\dagger D_s + b_k')\chi_k' = \Phi. \tag{67}$$

We can use the multi-mass solver; see Appendix A.

49

6 Maximal SYM with RHMC

Sec. 6.1 is for lattice people who knows almost nothing about SYM. String theorists, formal QFT people and lattice theorists who already know SYM can skip it.

6.1 The theories

Supersymmetry relates bosons and fermions. Because it changes the spin of the fields, if there are too many supercharges (i.e. generators of supersymmetry transformation) higher spin fields have to be involved. In order to construct a gauge theory without gravity, we cannot have massless fields with spin larger than one. It restricts the possible number of supercharges: the maximal number is sixteen. The *maximally supersymmetric Yang-Mills theories* (or *maximal* SYM) are the supersymmetric generalizations of Yang-Mills theory with sixteen supercharges. They can be obtained from SYM in (9+1) dimensions as follows.

Firstly, why do we care about $(9+1)$ dimensions? It is related to sixteen. In $(9+1)$-dimensional Minkowski space, there is a sixteen-dimensional spinor representation, which is Majorana and Weyl. Hence the minimal supersymmery transformation has sixteen components; and as we have seen above, it is maximal as well. The field content is simple: gauge field $A_M (M = 0, 1, \cdots, 9)$ and its superpartner (*gaugino*) $\psi_\alpha (\alpha = 1, 2, \cdots, 16)$. Both of them are $N \times N$ Hermitian matrices. The action is[21]

$$S_{(9+1)\text{-d}} = \frac{1}{g_{YM}^2} \int d^{10}x \text{Tr} \left(-\frac{1}{4} F_{MM'}^2 + \frac{i}{2} \bar{\psi} \gamma^M D_M \psi \right), \tag{68}$$

where $F_{MM'} = \partial_M A_{M'} - \partial_{M'} A_M + i[A_M, A_{M'}]$ and $D_M \psi = \partial_M \psi + i[A_M, \psi]$. γ_M is the left-handed part of the 10d gamma matrices, which can be chosen as

$$\begin{aligned}
\gamma^0 &= \mathbf{1} \otimes \mathbf{1} \otimes \mathbf{1} \otimes \mathbf{1}, \\
\gamma^1 &= \sigma_3 \otimes \mathbf{1} \otimes \mathbf{1} \otimes \mathbf{1}, \\
\gamma^2 &= \sigma_2 \otimes \sigma_2 \otimes \sigma_2 \otimes \sigma_2, \\
\gamma^3 &= \sigma_2 \otimes \sigma_2 \otimes \mathbf{1} \otimes \sigma_1, \\
\gamma^4 &= \sigma_2 \otimes \sigma_2 \otimes \mathbf{1} \otimes \sigma_3, \\
\gamma^5 &= \sigma_2 \otimes \sigma_1 \otimes \sigma_2 \otimes \mathbf{1}, \\
\gamma^6 &= \sigma_2 \otimes \sigma_3 \otimes \sigma_2 \otimes \mathbf{1}, \\
\gamma^7 &= \sigma_2 \otimes \mathbf{1} \otimes \sigma_1 \otimes \sigma_2, \\
\gamma^8 &= \sigma_2 \otimes \mathbf{1} \otimes \sigma_3 \otimes \sigma_2, \\
\gamma^9 &= \sigma_1 \otimes \mathbf{1} \otimes \mathbf{1} \otimes \mathbf{1}.
\end{aligned} \tag{69}$$

They are all real and symmetric. The fermion ψ is Majorana-Weyl, namely $\psi_\alpha^\dagger = \psi_\alpha$.

---

[21]We use $(-, +, \cdots, +)$ signature.

Theories in lower spacetime dimensions can be obtained by the dimensional reduction. Let us restrict spacetime to be $(p+1)$-dimensional. Namely, we do not allow the fields to depend on $x_{p+1}, \cdots, x_9$:

$$
\begin{aligned}
A_M(x_0, x_1, \cdots, x_p, \cdots, x_9) &= A_M(x_0, x_1, \cdots, x_p), \\
\psi_\alpha(x_0, x_1, \cdots, x_p, \cdots, x_9) &= \psi_\alpha(x_0, x_1, \cdots, x_p).
\end{aligned}
\tag{70}
$$

Let us use $\mu, \nu, \cdots$ and $I, J \cdots$ to denote $0, 1, \cdots, p$ and $p+1, \cdots, 9$. Also let us use X_I to denote A_I. Then

$$
\begin{aligned}
F_{\mu\nu} &= \partial_\mu A_\nu - \partial_\nu A_\mu + i[A_\mu, A_\nu], \\
F_{\mu I} &= \partial_\mu X_I - \partial_I A_\mu + i[A_\mu, X_I] = \partial_\mu X_I + i[A_\mu, X_I] = D_\mu X_I, \\
F_{IJ} &= \partial_I X_J - \partial_J X_I + i[X_I, X_J] = i[X_I, X_J].
\end{aligned}
\tag{71}
$$

Here $D_\mu X_I$ is the covariant derivative; X_I behaves as an adjoint scalar after the dimensional reduction. Hence the dimensionally reduced theory has the following action:

$$
S_{(p+1)\text{-d}} = S_B + S_F,
\tag{72}
$$

where

$$
S_B = \frac{1}{g_{YM}^2} \int d^{p+1}x \operatorname{Tr} \left(-\frac{1}{4} F_{\mu\nu}^2 - \frac{1}{2}(D_\mu X_I)^2 + \frac{1}{4}[X_I, X_J]^2 \right)
\tag{73}
$$

and

$$
S_F = \frac{1}{g_{YM}^2} \int d^{p+1}x \operatorname{Tr} \left(\frac{i}{2} \bar\psi \Gamma^\mu D_\mu \psi - \frac{1}{2} \bar\psi \Gamma^I [X_I, \psi] \right).
\tag{74}
$$

The Euclidean theory is obtained by performing the Wick rotation:

$$
\begin{aligned}
S_{\text{Euclidean}} = \ &\frac{1}{g_{YM}^2} \int d^{p+1}x \operatorname{Tr} \left(\frac{1}{4} F_{\mu\nu}^2 + \frac{1}{2}(D_\mu X_I)^2 - \frac{1}{4}[X_I, X_J]^2 \right. \\
&\left. + \frac{1}{2} \bar\psi D_t \psi + \frac{i}{2} \sum_{\mu=1}^{3} \bar\psi \gamma^\mu D_\mu \psi - \frac{1}{2} \bar\psi \Gamma^I [X_I, \psi] \right).
\end{aligned}
\tag{75}
$$

6.2 RHMC for SYM

By integrating out the fermions by hand, we obtain the Pfaffian[22] of the Dirac operator:

$$
Z = \int [dA][dX](\operatorname{Pf} D[A, X]) \cdot e^{-S_B[A,X]}.
\tag{76}
$$

---

[22]Pfaffian is defined for $2n \times 2n$ antisymmetric matrices M, where $n = 1, 2, \cdots$. Roughly speaking, the Pfaffian is the square root of the determinant: $(\operatorname{Pf} M)^2 = \det M$. More precisely, $\operatorname{Pf} M = \frac{1}{2^n n!} \sum_{\sigma \in S_{2n}} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} M_{\sigma(2i-1)\sigma(2i)}$.

Unfortunately, $\mathrm{Pf}D$ is not positive definite. Hence we use the absolute value of the pfaffian instead:

$$Z_{\text{phase quench}} = \int [dA][dX] |\mathrm{Pf}D[A, X]| \cdot e^{-S_B[A,X]}. \tag{77}$$

For the justification of this *phase quenching*, see Sec. 7.2.

In order to evaluate this integral, we use the RHMC algorithm.[23] The starting point is to rewrite $|\mathrm{Pf}\mathcal{D}| = \left(\det(D^\dagger D)\right)^{1/4}$ as

$$|\mathrm{Pf}D| = \int dF dF^* \exp\left(-((D^\dagger D)^{-1/8}F)^\dagger((D^\dagger D)^{-1/8}F)\right). \tag{78}$$

If we define Φ by $\Phi = (D^\dagger D)^{-1/8}F$, then Φ can be generated by the Gaussian weight $e^{-\Phi^\dagger \Phi}$, and F can be obtained by $F = \mathcal{D}^{1/8}\Phi$.

We further rewrite this expression by using the rational approximation

$$x^{-1/4} \simeq a_0 + \sum_{k=1}^{Q} \frac{a_k}{x + b_k}. \tag{79}$$

Then we can replace the Pfaffian with

$$|\mathrm{Pf}D| = \int dF dF^* \exp\left(-S_{\text{PF}}\right), \tag{80}$$

where

$$S_{\text{PF}} = a_0 F^\dagger F + \sum_{k=1}^{Q} a_k F^\dagger (D^\dagger D + b_k)^{-1} F. \tag{81}$$

The parameters a_k, b_k, a_k' and b_k' are real and positive, and can be chosen so that the approximation is sufficiently good within the range of $D^\dagger D$ during the simulation.

Now it is clear that everything is the same as RHMC for strange quark except that the powers are different. Therefore, the molecular evolution goes as follows:

1. Generate Φ by the Gaussian weight.

2. Calculate $F = (D^\dagger D)^{1/8}\Phi$.

3. Calculate H_i.

4. Fix F and update A_μ and X_I.

5. Calculate H_f.

---

[23]RHMC has been widely used in lattice QCD. The first application to SYM can be found in [18].

6. Do the Metropois test.

Computationally most demanding part is solving the linear equations

$$(\mathcal{D} + b_k)\chi_k = F \quad (k = 1, \cdots, Q), \tag{82}$$

which appears in the derivative of S_{PF},

$$\frac{\partial S_{PF}}{\partial A} = -\sum_{k=1}^{Q} a_k \chi_k^\dagger \frac{\partial \mathcal{D}}{\partial A} \chi_k, \qquad \frac{\partial S_{PF}}{\partial X} = -\sum_{k=1}^{Q} a_k \chi_k^\dagger \frac{\partial \mathcal{D}}{\partial X} \chi_k. \tag{83}$$

This is much easier than evaluating the Pfaffian.

7 Difference between SYM and QCD

Let us explain important differences between SYM and QCD simulations.[24]

7.1 Parameter fine tuning problem and its cure

Symmetry plays important roles in physics. For example, why is the pion so light? Because QCD has approximate chiral symmetry and pion is the Nambu-Goldstone boson associated with this symmetry. Finite quark mass breaks chiral symmetry softly, so that a light mass of pion is generated. The Wilson fermion breaks chiral symmetry explicitly. If we use it as a lattice regularization, the radiative corrections generate pion mass, and hence we have to fine-tune the bare quark mass to keep the pion light. This is a well known example of *the parameter fine tuning problem.* This problem does not exist if we use overlap fermion [19] or domain-wall fermion [20].

As another simple example, let us consider the plaquette action (36). It has many *exact symmetries* at *regularized level*:

- Gauge symmetry $U_{\mu,\vec{x}} \to \Omega_{\vec{x}} U_{\mu,\vec{x}} \Omega_{\vec{x}+\hat{\mu}}^\dagger$, where $\Omega_{\vec{x}}$ are unitary matrices defined on sites.

- Discrete translation $\vec{x} \to \vec{x} + \hat{\mu}$

- 90-degree rotations, e. g. $\hat{x} \to \hat{y}, \hat{y} \to -\hat{x}$

- Parity $\hat{x}, \hat{y}, \hat{z} \to -\hat{x}, -\hat{y}, -\hat{z}$

- Charge conjugation $U \to U^\dagger$

---

[24]This section has large overlap with other review articles I have written in the past, but I include it here in order to make this article self-contained.

The integral measure is taken to be the Haar measure, which respects these symmetries. Therefore the radiative corrections cannot break them: these are the symmetries of the quantum theory.[25] What happens if we use a regulator which breaks these symmetries, say a momentum cutoff which spoils gauge symmetry? Then we need counter terms, whose coefficients are fine tuned, to restore the gauge invariance in the continuum.

So now we know the basic strategy for the lattice simulation of supersymmetric theories — let's keep supersymmetry exactly on a lattice! But there is a one-sentence proof of a 'no-go theorem': because supersymmetry algebra contains infinitesimal translation $\{Q_\alpha, \bar{Q}_\beta\} \sim \gamma^\mu P_\mu$, which is explicitly broken on any lattice by construction, it is impossible to keep entire algebra exactly.

Still, it is not the end of the game. Exact symmetry at the regularized level is sufficient, but it is not always necessary. Sometimes the radiative corrections can be controlled by using some other symmetries or a part of supersymmetry algebra.

The first breakthrough emerged for 4d $\mathcal{N} = 1$ pure SYM. For this theory, the supersymmetric continuum limit is realized if the gaugino mass is set to zero. This can be achieved if the chiral symmetry is realized on lattice [21, 22]. Of course it was not really a 'solution' at that time, due to the Nielsen-Ninomiya no-go theorem [23]. Obviously, that the author of [21] proposed a way to circumvent the no-go theorem [20] was not a coincidence. (Note that the one-parameter fine tuning can be tractable in this case. See [24] for recent developments.)

The second breakthrough was directly motivated by the gauge/gravity duality [25].[26] The duality relates supersymmetric theories in various spacetime dimensions, including the ones with less than four dimensions, to string/M-theory [27]. Some prior proposals [6, 7, 29, 30] related lower dimensional theories to quantum gravity as well. Because Yang-Mills in less than four dimensions are super-renormalizable, possible radiative corrections are rather constrained. In particular, in two dimensions, supersymmetric continuum limit can be guaranteed to all order in perturbation theory by keeping some supersymmetries in a clever manner [26].

Yet another breakthrough came from superstring theory. Maximally supersymmetric Yang-Mills in $(p + 1)$ dimensions describe the low-energy dynamics of Dp-branes [31]. Furthermore higher dimensional D-branes can be made by collecting lower dimensional D-branes [32]. In terms of gauge theory, higher dimensional theories can be obtained as specific vacua of lower dimensional large-N theories. Or in other words, spacetime emerges from matrix degrees of freedom [33].

Based on these ideas, various regularization methods have been invented. A list of regularization schemes for 4-, 8- and 16-SUSY theories without additional matter (i.e. dimensional reductions of 4d, 6d and 10d $\mathcal{N} = 1$ pure super Yang-Mills) is shown below.

---

[25]In this case, discrete translation and rotation symmetries guarantee the invariance under continuous transformations in the continuum limit.

[26]D. B. Kaplan and M. Ünsal told me that their biggest motivation was in the gauge/gravity duality. For several years this motivation has not been widely shared among the followers, probably because they did not mention it explicitly in their first paper [26].

For more details see [34] and references therein.

Dimensions	# of SUSY	Regularization scheme	Fine tuning
0+1	4,8,16	lattice	none
0+1	4,8,16	momentum cutoff	none
1+1	4,8,16	lattice	none
2+1	4,8,16	lattice	necessary
2+1	4,8 (large-N)	fuzzy sphere	none
2+1	16	fuzzy sphere	none
3+1	4	lattice	none
3+1	8,16	lattice	necessary
3+1	8 (large-N)	lattice + fuzzy sphere	none
3+1	16	lattice + fuzzy sphere	none
3+1	4,8,16 (large-N)	Eguchi-Kawai reduction	none

7.2 Sign problem and its cure

A crucial assumption of MCMC is that the path-integral weight e^{-S} is real and positive; otherwise it is not 'probability'. This condition is often broken. The simplest example is the path-integral in Minkowski space — the weight e^{iS}, where S is real, is complex. One may think this problem is gone in Euclidean space; the life is not that easy, the action S can be complex after the Wick rotation; it only has to satisfy the reflection positivity, which is the counterpart of the reality in Minkowski space. Well-known evils which cause the sign problem include the θ-term and the Chern-Simons term in Yang-Mills theory, and QCD with finite baryon chemical potential. Note that, although it is often called 'fermion sign problem', the source of the sign is not necessary the fermionic part.

Maximal super Yang-Mills have the sign problem, because the pfaffian can take complex values.

There is no known generic solution of the sign problem. It has been argued that the sign problem is NP-hard [35], and hence, unless you have a reason to think P=NP, it is a waste of time to invest your time in searching for a generic solution. However, theory-specific solutions are not excluded. Probably the best case scenario is that somebody comes up with a clever change of variables so that the sign problem disappears. I actually had such a lucky experience when he studied the ABJM matrix model in Ref. [4], thanks to smart students who knew important analytic formulas developed in [36]. Certain supersymmetric theories seem to allow yet another beautiful solution, as we will explain below.

7.2.1 Phase reweighting

Let us consider SYM as a concrete example. The expectation value of an observable $\mathcal{O}$, which is written in terms of A_μ and X_I, with the 'full' partition function (76) is

$$\langle \mathcal{O}(A, X) \rangle_{\text{full}} = \frac{\int [dA][dX](\text{Pf}D[A, X]) \cdot e^{-S_B[A,X]} \cdot \mathcal{O}(A, X)}{\int [dA][dX](\text{Pf}D[A, X]) \cdot e^{-S_B[A,X]}}. \tag{84}$$

By writing $\mathrm{Pf}D = |\mathrm{Pf}D|e^{i\theta}$, we can easily see

$$\langle \mathcal{O}(A, X) \rangle_{\text{full}} = \frac{\langle \mathcal{O}(A, X) \cdot e^{i\theta[A,X]} \rangle_{\text{phase quench}}}{\langle e^{i\theta[A,X]} \rangle_{\text{phase quench}}}, \tag{85}$$

where $\langle \cdot \rangle_{\text{phase quench}}$ is the expectation value with the phase quenched path-integral (77). By using this trivial identity, in principle we can calculate the expectation value in the full theory. This method is called *phase reweighting*.

However there are several reasons we do not want to use the phase reweighting method. Firstly, calculation of the pfaffian is numerically very demanding, and although we can avoid explicit evaluation of the Pfaffian in the configuration generation with the phase quenched ensemble, in order to calculate (85) we must calculate the Pfaffian explicitly. Secondly, if the phase factor $e^{i\theta}$ fluctuates rapidly, both the denominator and numerator becomes very small, and numerically it is difficult to distinguish them from zero. Then the expression is practically 0/0, whose error bar is infinitely large. The fluctuation is particularly violent when the full theory and phase quenched theory does not have substantial overlap (see Sec. 3.3.1), and it actually happens in finite density QCD. As we will argue shortly, the full theory and phase quenched theory are rather close in the case of SYM. Still, the phase fluctuation is very large and it is impossible to calculate the average phase at large N and/or large volume. Roughly speaking, θ is the imaginary part of the action, which scales as N^2 times volume!

Although the phase reweighting is an unrealistic approach to SYM, it can be a practical tool when the simulation cost is cheaper. For example it is a common strategy in certain area of condensed matter physics.

7.2.2 Phase quench

Let us consider the phase quench approximation. Namely, we calculate $\langle \mathcal{O}(A, X) \rangle_{\text{phase quench}}$, which does not take into account the phase at all, instead of $\langle \mathcal{O}(A, X) \rangle_{\text{full}}$. Such an 'approximation' can make sense when $\mathcal{O}$ and $e^{i\theta}$ factorize, $\langle \mathcal{O}e^{i\theta} \rangle_{\text{phase quench}} \simeq \langle \mathcal{O} \rangle_{\text{phase quench}} \times \langle e^{i\theta} \rangle_{\text{phase quench}}$.

The phase quench approximation almost always fails. Very fortunately, rare exceptions include SYM. Firstly, at high temperature and not so large N, θ is close to zero and hence the phase can safely be ignored. Rather surprisingly, θ remains small down to rather low temperature and moderately large N, which is relevant for testing the gauge/gravity duality; see e.g. [37, 38, 39, 40, 41] for explicit checks for $(0+1)$- and $(1+1)$-dimensional theories. At low temperature or with supersymmetric boundary condition, the phase factor does oscillate. Still, as long as the fluctuation is not very rapid, we can defeat the sign by brute force. Then we can confirm $\langle \mathcal{O} \rangle_{\text{phase quench}} = \langle \mathcal{O} \rangle_{\text{full}}$ within numerical error, for some observables [40, 42]. When the phase factor fluctuates rapidly, or the dimension of the Dirac operator is too big, we cannot evaluate the effect of the phase. Still we can make an indirect argument based on numerical evidence that the phase quench approximation is good for certain observables [43].

Note that the phase quench leads to a wrong result for the type IIB matrix model [28], which is the dimensional reduction of maximal SYM to zero dimension.

7.3 Flat direction and its cure

Many supersymmetric Yang-Mills theories (practically all theories relevant in the context of the gauge/gravity duality) have adjoint scalar fields X_I. There are flat directions along which scalar matrices commute, $[X_I, X_J] = 0$, and the eigenvalues roll to infinity. With 2 or 3 noncompact spatial dimensions, we can fix the eigenvalues by hand; it is a choice of moduli, due to the superselection. However with compact space[27] or lower dimensions the eigenvalue distribution should be determined dynamically. There is nothing wrong with this: the flat direction is a feature of the theory, it is not a bug. However it causes headache when we perform MCMC for two reasons: firstly, the partition function is not convergent, and hence the simulation never thermalizes [37]. Secondly, when the eigenvalues roll too far, the lattice simulation runs into an unphysical lattice artifact [44].

In order to tame the flat directions, we have to understand its physical meaning. $(p+1)$-d U(N) SYM describe a system of N Dp-branes sitting parallel to each other, and the eigenvalues $(X_1^{ii}, X_2^{ii}, \cdots, X_{9-p}^{ii})$ describe the location of the i-th Dp-brane. When eigenvalues form a bound state, it can be regarded as a black p-brane. When one of the eigenvalues is separated far from others, the interaction is very weak due to supersymmetry; the potential is proportional to $-f(T)/r^{8-p}$ [45, 46], where r is the distance, T is temperature of the black brane and $f(T)$ is a monotonically increasing function which vanishes at $T = 0$. This potential is not strong enough to trap a Dp-brane once it is emitted from the black brane. In the same manner, phases with multiple bunches of eigenvalues can exist; they describe multiple black branes. This is the reason that the flat directions exist; the partition function is not convergent because there are too many different classes of configurations.

Now our task is apparent: we should cut out configurations describing certain physical situation we are interested. The first thing we should study is a single black hole or black brane, namely a bound state of all eigenvalues. This bound state is only metastable, but as N increases it becomes stable enough so that we can collect sufficiently many configurations without seeing the emission of the eigenvalues [37]. Large-N behavior of the $(0 + 1)$-dimensional theory has been studied in this manner in [37, 38, 43]. When N is small, we need to introduce a cutoff for the eigenvalues. In [47] a cutoff is introduced for $(1/N) \sum_M \text{Tr} X_M^2$, and by carefully changing the cutoff the property of the single bunch phase has been extracted. It is also possible to introduce a mass term $Nm^2 \int d^{p+1}x \text{Tr} X_M^2$ and take $m \to 0$ [48, 44]. Note that one has to make sure that the flat direction is under control when m^2 is small, as demonstrated in [48, 44], because simulations often pick up the flat direction and end up in the U(1)^N vacuum. See also [49, 50][51, 52, 53, 41][54] for the details on how to control the flat directions in actual simulations. It is also possible to add slightly more complicated deformation which preserves (at least a part of) supersymmetry,

---

[27]More precisely, the torus compactification. For the compactification to a curved manifold, the flat directions are lifted by a mass term associated with the curvature.

which is so-called 'plane wave deformation' [55]. See [55, 56] for various $(0+1)$-d theories and [57, 58, 59] for $(1+1)$-d lattice construction. See also [60] for other cases.

8   Conclusion

In this article I have tried to explain the essence of Markov Chain Monte Carlo. Simple algorithms like the Metropolis algorithm is good enough for solving nontrivial problems in hep-th literature on a laptop.

Serious simulations of super Yang-Mills require more efforts, but the basic idea is the same. We just have to update the configurations more efficiently. For that, we can use efficient techniques developed in lattice QCD community: HMC, RHMC and many more which I haven't explained in this article. They can be implemented to our simulation code by using $+, -, \times, \div, \sin, \cos, \exp, \log, \sqrt{}$, "if" and loop. There are some challenging issues specific to supersymmetric theories, most notably the problem associated with flat directions, but we already know basic strategies to handle them.

In this article I did not explain the complete details of lattice SYM simulation. The detail of the simulations of BFSS matrix model can be found in the article available at [8].

Acknowledgement

A   Multi-mass CG method

A.1   (Single-mass) CG method

Let us first introduce the ordinary (or 'single-mass') conjugate gradient method. The notation is that of [61]. We want to solve the linear equation

$$A\vec{x} = \vec{b}, \tag{86}$$

where $A = D^\dagger D$. We construct a sequence of approximate solutions $\vec{x}_1, \vec{x}_2, \cdots$ which (almost always) converges to the solution.

We start with an initial trial solution $\vec{x}_1$, which is arbitrary. From this we define $\vec{r}_1, \vec{\tilde{r}}_1, \vec{p}_1, \vec{\tilde{p}}_1$ as

$$\vec{r}_1 = \vec{\tilde{r}}_1 = \vec{p}_1 = \vec{\tilde{p}}_1 = \vec{b} - A\vec{x}_1. \tag{87}$$

Then, we construct $\vec{x}_k$ as follows:

1. $\alpha_k = \frac{\vec{r}_k^\dagger \cdot \vec{r}_k}{\vec{p}_k^\dagger \cdot A \vec{p}_k}$.

2. $\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{p}_k$.

3. $\vec{r}_{k+1} = \vec{r}_k - \alpha_k A \vec{p}_k$.

4. $\beta_k = \frac{\vec{r}_{k+1}^\dagger \cdot \vec{r}_{k+1}}{\vec{r}_k^\dagger \cdot \vec{r}_k}$.

5. $\vec{p}_{k+1} = \vec{r}_{k+1} + \beta_k \vec{p}_k$.

Note that $\vec{r}_k = \vec{b} - A\vec{x}_k$. The norm of $\vec{r}_k$ converges to zero, or equivalently, $\vec{x}_k$ converges to the solution.

A.2 Multi-mass CG method

Let A_σ be 'shifted' version of A:

$$A_\sigma = A + \sigma \cdot \mathbf{1}. \tag{88}$$

If A were the Laplacian, σ would be interpreted as a mass term. The multi-mass CG solver [16] enables us to solve $A_\sigma \vec{x} = \vec{b}$ for many different values of σ simultaneously, with only negligibly small additional cost.

The key idea is that, from the iterative series for A,

$$\begin{aligned}
\vec{r}_{k+1} &= \vec{r}_k - \alpha_k A \vec{p}_k, \\
\vec{p}_{k+1} &= \vec{r}_{k+1} + \beta_k \vec{p}_k,
\end{aligned} \tag{89}$$

it is possible to construct a similar series for the shifted operator,

$$\begin{aligned}
\vec{r}_{k+1}^\sigma &= \vec{r}_k^\sigma - \alpha_k^\sigma A_\sigma \vec{p}_k^\sigma, \\
\vec{p}_{k+1}^\sigma &= \vec{r}_{k+1}^\sigma + \beta_k^\sigma \vec{p}_k^\sigma,
\end{aligned} \tag{90}$$

where

$$\vec{r}_k^\sigma = \zeta_k^\sigma \vec{r}_k. \tag{91}$$

Indeed, (90) can be satisfied by taking[28]

$$
\begin{aligned}
\alpha_k^\sigma &= \alpha_k \cdot \frac{\zeta_{k+1}^\sigma}{\zeta_k^\sigma}, \\
\beta_k^\sigma &= \beta_k \cdot \left(\frac{\zeta_{k+1}^\sigma}{\zeta_k^\sigma}\right)^2, \\
\zeta_{k+1}^\sigma &= \frac{\zeta_k^\sigma \zeta_{k-1}^\sigma \alpha_{k-1}}{\alpha_{k-1}\zeta_{k-1}^\sigma(1+\alpha_k\sigma) + \alpha_k\beta_{k-1}(\zeta_{k-1}^\sigma - \zeta_k^\sigma)}.
\end{aligned}
\tag{93}
$$

Therefore, we can generalize the usual BiCG solver in the following manner:

1. $\alpha_k = \frac{\vec{r}_k^\dagger \cdot \vec{r}_k}{\vec{p}_k^\dagger \cdot A\vec{p}_k}$.

2. Calculate ζ_{k+1}^σ and α_k^σ using (93).

3. $\vec{x}_{k+1}^\sigma = \vec{x}_k^\sigma + \alpha_k^\sigma \vec{p}_k^\sigma$.

4. $\vec{r}_{k+1} = \vec{r}_k - \alpha_k A\vec{p}_k$.

5. $\beta_k = \frac{\vec{r}_{k+1}^\dagger \cdot \vec{r}_{k+1}}{\vec{r}_k^\dagger \cdot \vec{r}_k}$.

6. Calculate β_k^σ using (93).

7. $\vec{p}_{k+1} = \vec{r}_{k+1} + \beta_k \vec{p}_k$,
   $\vec{p}_{k+1}^\sigma = \vec{r}_{k+1}^\sigma + \beta_k^\sigma \vec{p}_k^\sigma$.

Then, $\vec{x}_k^\sigma$ is an approximate solution with the residual vector $\vec{r}_k^\sigma$,

$$
\vec{r}_k^\sigma = \vec{b} - A_\sigma \vec{x}_k^\sigma.
\tag{94}
$$

Note that we cannot start with an arbitrary initial condition, because (90) is not satisfied then. We choose the following special initial condition in order to satisfy (90):

$$
\begin{aligned}
\vec{x}_1 &= \vec{x}_1^\sigma = \vec{p}_0 = \vec{p}_0^\sigma = \vec{0}, \\
\vec{r}_1 &= \vec{r}_1^\sigma = \vec{r}_0 = \vec{r}_0^\sigma = \vec{p}_1 = \vec{p}_1^\sigma = \vec{b}, \\
\zeta_0^\sigma &= \zeta_1^\sigma = \alpha_0 = \alpha_0^\sigma = \beta_0 = \beta_0^\sigma = 1.
\end{aligned}
\tag{95}
$$

As long as we stick to this initial condition, it is hard to implement any preconditioning. If you know any preconditioning which can be used with multi-mass CG solver, please let us know.

---

[28]From (89) we have

$$
\vec{r}_{k+1} = \left(1 + \frac{\alpha_k\beta_{k-1}}{\alpha_{k-1}}\right)\vec{r}_k - \alpha_k A\vec{r}_k - \frac{\alpha_k\beta_{k-1}}{\alpha_{k-1}}\vec{r}_{k-1}.
\tag{92}
$$

Comparing the coefficients with those in a similar equation obtained from (90), we obtain (93).

B   Box-Muller method

Let p and q be uniform random numbers in $[0, 1]$. There are many random number generators which give you such p and q. Then, $x = \sqrt{-2\log p}\sin(2\pi q)$ and $y = \sqrt{-2\log p}\cos(2\pi q)$ are random numbers with weights $\frac{e^{-x^2/2}}{\sqrt{2\pi}}$ and $\frac{e^{-y^2/2}}{\sqrt{2\pi}}$.

C   Jackknife method: generic case

In Sec. 3.2.1, we assumed that a quantity of interest can be calculated for each sample. Let us consider more generic cases; for example in order to determine the mass of particle excitation we need to calculate two-point function by using many samples and then extract the mass from that, and hence the mass cannot be calculated sample by sample.

In the Jackknife method, we first divide the configurations to bins with width w; the first bin is $\{x^{(1)}\}, \{x^{(2)}\}, \cdots, \{x^{(w)}\}$, the second bin is $\{x^{(w+1)}\}, \{x^{(w+2)}\}, \cdots, \{x^{(2w)}\}$, etc. Suppose we have n bins. Then we define the average of an observable $f(x)$ with k-th bin removed,

$$\overline{f}^{(k,w)} \equiv (\text{the value calculated after removing} k\text{-th bin}). \tag{96}$$

The average value is defined by

$$\overline{f} \equiv \frac{1}{n}\sum_k \overline{f}^{(k,w)}. \tag{97}$$

The Jackknife error is defined by

$$\Delta_w \equiv \sqrt{\frac{n-1}{n}\sum_k \left(\overline{f}^{(k,w)} - \overline{f}\right)^2}. \tag{98}$$

References

[1] J. Penedones, private communication.

[2] T. Azeyanagi, M. Hanada, T. Hirata and T. Ishikawa, JHEP **0801**, 025 (2008) [arXiv:0711.1925 [hep-lat]].

[3] T. Azeyanagi, M. Hanada, M. Unsal and R. Yacoby, Phys. Rev. D **82**, 125013 (2010) [arXiv:1006.0717 [hep-th]].

[4] M. Hanada, M. Honda, Y. Honma, J. Nishimura, S. Shiba and Y. Yoshida, JHEP **1205**, 121 (2012) doi:10.1007/JHEP05(2012)121 [arXiv:1202.5300 [hep-th]].

[5] http://www.netlib.org/lapack/

[6] T. Banks, W. Fischler, S. H. Shenker and L. Susskind, Phys. Rev. D **55**, 5112 (1997) [hep-th/9610043].

[7] B. de Wit, J. Hoppe and H. Nicolai, Nucl. Phys. B **305**, 545 (1988).

[8] M. Hanada, A simulation code written for Monte Carlo String/M-theory Collaboration, freely available at `https://sites.google.com/site/hanadamasanori/home/mmmm`

[9] D. Schaich and T. DeGrand, Comput. Phys. Commun. **190**, 200 (2015) [arXiv:1410.6971 [hep-lat]].

[10] D. Schaich, Codes for supersymmetric lattice gauge theories , https://github.com/daschaich/susy

[11] S. Catterall and A. Joseph, Comput. Phys. Commun. **183**, 1336 (2012) [arXiv:1108.1503 [hep-lat]].

[12] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953). Equation of state calculations by fast computing machines. The journal of chemical physics, 21(6), 1087-1092.

[13] O. Aharony, O. Bergman, D. L. Jafferis and J. Maldacena, JHEP **0810**, 091 (2008) [arXiv:0806.1218 [hep-th]].

[14] S. Duane, A. D. Kennedy, B. J. Pendleton and D. Roweth, Phys. Lett. B **195**, 216 (1987).

[15] M. A. Clark and A. D. Kennedy, "The RHMC algorithm for 2 flavors of dynamical staggered fermions," Nucl. Phys. Proc. Suppl. **129**, 850 (2004) [arXiv:hep-lat/0309084].

[16] B. Jegerlehner, "Krylov space solvers for shifted linear systems," arXiv:hep-lat/9612014.

[17] M.A. Clark and A.D. Kennedy, https://github.com/maddyscientist/AlgRemez

[18] S. Catterall and T. Wiseman, JHEP **0712**, 104 (2007) [arXiv:0706.3518 [hep-lat]].

[19] H. Neuberger, Phys. Lett. B **417**, 141 (1998) [hep-lat/9707022].

[20] D. B. Kaplan, Phys. Lett. B **288**, 342 (1992) [hep-lat/9206013].

[21] D. B. Kaplan, Phys. Lett. B **136**, 162 (1984).

[22] G. Curci and G. Veneziano, Nucl. Phys. B **292**, 555 (1987).

[23] H. B. Nielsen and M. Ninomiya, Nucl. Phys. B **185**, 20 (1981) Erratum: [Nucl. Phys. B **195**, 541 (1982)]. doi:10.1016/0550-3213(81)90361-8, 10.1016/0550-3213(82)90011-6

[24] S. Ali, G. Bergner, H. Gerber, P. Giudice, I. Montvay, G. Mnster, S. Piemonte and P. Scior, JHEP **1803**, 113 (2018) [arXiv:1801.08062 [hep-lat]].

[25] J. M. Maldacena, Int. J. Theor. Phys. **38**, 1113 (1999) [Adv. Theor. Math. Phys. **2**, 231 (1998)] [hep-th/9711200].

[26] D. B. Kaplan, E. Katz and M. Unsal, JHEP **0305**, 037 (2003) [hep-lat/0206019].

[27] N. Itzhaki, J. M. Maldacena, J. Sonnenschein and S. Yankielowicz, Phys. Rev. D **58**, 046004 (1998) [hep-th/9802042].

[28] N. Ishibashi, H. Kawai, Y. Kitazawa and A. Tsuchiya, Nucl. Phys. B **498**, 467 (1997) [hep-th/9612115].

[29] L. Motl, hep-th/9701025.

[30] R. Dijkgraaf, E. P. Verlinde and H. L. Verlinde, Nucl. Phys. B **500**, 43 (1997) [hep-th/9703030].

[31] E. Witten, Nucl. Phys. B **460**, 335 (1996) [hep-th/9510135].

[32] R. C. Myers, JHEP **9912**, 022 (1999) [hep-th/9910053].

[33] Probably the first demonstration of this philosophy was in T. Eguchi and H. Kawai, Phys. Rev. Lett. **48**, 1063 (1982).

[34] M. Hanada, Int. J. Mod. Phys. A **31**, no. 22, 1643006 (2016) [arXiv:1604.05421 [hep-lat]].

[35] M. Troyer and U. J. Wiese, Phys. Rev. Lett. **94**, 170201 (2005) [cond-mat/0408370].

[36] A. Kapustin, B. Willett and I. Yaakov, JHEP **1010**, 013 (2010) [arXiv:1003.5694 [hep-th]].

[37] K. N. Anagnostopoulos, M. Hanada, J. Nishimura and S. Takeuchi, "Monte Carlo studies of supersymmetric matrix quantum mechanics with sixteen supercharges at finite temperature," Phys. Rev. Lett. **100**, 021601 (2008) [arXiv:0707.4454 [hep-th]].

[38] M. Hanada, Y. Hyakutake, J. Nishimura and S. Takeuchi, Phys. Rev. Lett. **102**, 191602 (2009) [arXiv:0811.3102 [hep-th]].

[39]

[39] V. G. Filev and D. O'Connor, "The BFSS model on the lattice," arXiv:1506.01366 [hep-th].

[40] M. Buchoff, M. Hanada and S. Matsuura, unpublished.

[41] S. Catterall, R. G. Jha, D. Schaich and T. Wiseman, arXiv:1709.07025 [hep-th].

[42] M. Hanada, J. Nishimura, Y. Sekino and T. Yoneya, JHEP **1112**, 020 (2011) [arXiv:1108.5153 [hep-th]].

[43] E. Berkowitz, E. Rinaldi, M. Hanada, G. Ishiki, S. Shimasaki and P. Vranas, Phys. Rev. D **94**, no. 9, 094501 (2016) [arXiv:1606.04951 [hep-lat]].

[44] M. Hanada and I. Kanamori, JHEP **1101**, 058 (2011) [arXiv:1010.2948 [hep-lat]].

[45] U. H. Danielsson, G. Ferretti and B. Sundborg, Int. J. Mod. Phys. A **11**, 5463 (1996) [hep-th/9603081].

[46] D. N. Kabat and P. Pouliot, Phys. Rev. Lett. **77**, 1004 (1996) [hep-th/9603127].

[47] M. Hanada, Y. Hyakutake, G. Ishiki and J. Nishimura, Science **344**, 882 (2014) [arXiv:1311.5607 [hep-th]].

[48] M. Hanada and I. Kanamori, Phys. Rev. D **80**, 065014 (2009) [arXiv:0907.4966 [hep-lat]].

[49] S. Catterall, A. Joseph and T. Wiseman, JHEP **1012**, 022 (2010) [arXiv:1008.4964 [hep-th]].

[50] E. Gigure and D. Kadoh, JHEP **1505**, 082 (2015) [arXiv:1503.04416 [hep-lat]].

[51] S. Catterall, P. H. Damgaard, T. Degrand, R. Galvez and D. Mehta, JHEP **1211**, 072 (2012) [arXiv:1209.5285 [hep-lat]].

[52] S. Catterall, D. Schaich, P. H. Damgaard, T. DeGrand and J. Giedt, Phys. Rev. D **90**, no. 6, 065013 (2014) [arXiv:1405.0644 [hep-lat]].

[53] S. Catterall and D. Schaich, JHEP **1507**, 057 (2015) [arXiv:1505.03135 [hep-lat]].

[54] D. August, B. H. Wellegehausen and A. Wipf, arXiv:1802.07797 [hep-lat].

[55] D. E. Berenstein, J. M. Maldacena and H. S. Nastase, JHEP **0204**, 013 (2002) [hep-th/0202021].

[56] N. Kim and J. H. Park, Nucl. Phys. B **759**, 249 (2006) [hep-th/0607005].

[57] M. Hanada, S. Matsuura and F. Sugino, Prog. Theor. Phys. **126**, 597 (2011) [arXiv:1004.5513 [hep-lat]].

[58] M. Hanada, JHEP **1011**, 112 (2010) [arXiv:1009.0901 [hep-lat]].

[59] M. Hanada, S. Matsuura and F. Sugino, Nucl. Phys. B **857**, 335 (2012) [arXiv:1109.6807 [hep-lat]].

[60] J. Kato, Y. Kondo and A. Miyake, JHEP **1109**, 019 (2011) [arXiv:1104.1252 [hep-th]].

[61] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling, "Numerical Recipes in FORTRAN, " Cambridge University Press.