Worksheet week 7 – Statistical physics on a computer

1 Purpose

1. Become familiar with the use of pseudo-random numbers in C++.
2. Explore the statistical physics of a small number of spins in a 1D non-interacting Ising model.
3. Implement a (poor) Markov chain Monte Carlo algorithm.

2 Random numbers in computational physics

2.1 Pseudo-random number generation

Random numbers are used ubiquitously throughout computational physics, and we will be relying on them to implement our Monte Carlo algorithms. Unfortunately, since computers are fundamentally deterministic, they can't be used to generate true random numbers. So to approximate randomness, we have to resort to using a *pseudo*-random number generator (PRNG). PRNGs are purpose-built algorithms that produce deterministic sequences of numbers that "seem" random[1].

Importantly, if you "seed" the random number generator with a particular starting point, it will always go on to produce the same sequence of numbers. This is actually a very useful feature for debugging code, because your "random" calculation will have the same output every time. However, if you want to run your code multiple times and have it produce different outputs each time, you need to remember to change the seed each time you run it. There are a number of common methods to set the seed so that it is automatically different every run: you can use the current time, or you can seed your PRNG with a random number from a different source (e.g. a "true" random number source).

C++11 has a built-in PRNG library for generating random numbers (documentation can be found at e.g. `https://cppreference.com/w/cpp/numeric/random`). This library contains a number of PRNG algorithms, as well as methods for generating pseudo-random numbers from various distributions (e.g. uniform or Gaussian), and in various types (e.g. integers between 1 and 10, or doubles between 0 and 1). It also provides facilities for generating "true" random numbers, via `std::random_device`.

The script 'randomExample.cpp' is provided as an example of how generate random numbers in C++. This script demonstrates the use of `std::random_device`, as well as the commonly used Mersenne Twister algorithm `std::mt19937`, for generating integers. For the purposes of this course, we will mostly need uniform random numbers between 0 and 1, so the most important function will be `std::uniform_real_distribution`.

2.2 Exercise – Estimating π

Monte Carlo methods originated as a way of estimating integrals that couldn't be solved analytically (for an interesting account of the history, see e.g. Chapter 1.4 of "Monte Carlo Methods in Statistical Physics" by Barkema and Newman). Fundamentally, a Monte Carlo method is simply a way of using repeated random sampling to obtain a result. The name "Monte Carlo" was coined by Nicholas Metropolis—it is a reference to the Monte Carlo casino in Monaco, and is meant to reflect the randomness that is inherent to the method.

[1] To check "how random" a given algorithm is, you can perform various "randomness tests" on their outputs. It turns out that a lot of widely used PRNGs actually fail many of these tests. We won't worry about that here, though. The C++ random number facilities are very high quality.

As a simple demonstration of how random numbers can be used to estimate integrals, we can use our random number generator to calculate the ratio between two areas: (i) the area of a square of sidelength L, and (ii) the area of a circle of diameter L. Of course, we know that this ratio should be equal to:

$$\frac{A_{\text{circle}}}{A_{\text{square}}} = \frac{\pi(L/2)^2}{L^2} = \pi/4. \tag{1}$$

In the Monte Carlo approach, we repeatedly sample random co-ordinates (x, y) uniformly from within the domain $-L/2 \leq x, y < L/2$. If we do this enough times, we will eventually build up a distribution of points that will evenly cover the square. The trick is to then measure the fraction of our randomly generated points that happen to fall within the circle's radius, i.e. points that satisfy $(x^2 + y^2)^{1/2} \leq L/2$. We should find that:

$$\frac{N_{\text{circle}}}{N_{\text{square}}} \approx \frac{A_{\text{circle}}}{A_{\text{square}}} = \pi/4. \tag{2}$$

So we can use this very simple approach as a way of estimating π. The more points (x, y) you generate, the more accurate your estimate of π will become.

Write some C++ code to perform this calculation (use the provided codes to help you implement the random number generation). Each time you generate a random pair (x, y), calculate the ratio in Eq. (2), and output the relative error $\epsilon \equiv |\pi_{\text{num}} - \pi|/\pi$, where $\pi_{\text{num}} = 4N_{\text{circle}}/N_{\text{square}}$. Save the output to a file, and make a plot of $\log(N_{\text{square}})$ vs. $\log(\epsilon)$. You should find that the data looks like a straight line, with slope $-1/2$. This tells us that the error scales as $\epsilon \sim 1/\sqrt{N_{\text{square}}}$, so if we want to reduce the error by a factor of two, we need to increase the number of samples by a factor of four. This scaling is ubiquitous, and shows up whenever discrete events are being counted.

3 Statistical physics of a spin chain

We are going to explore the thermodynamic properties of a small 1D chain of N non-interacting spins $\{\sigma_i\}$ in a uniform magnetic field B. We will describe these spins by a non-interacting Ising Hamiltonian of the form:

$$H = -\mu B \sum_i^N \sigma_i, \tag{3}$$

where μ is the magnetic moment of each spin, and $\sigma_i = +1$ (spin up) or -1 (spin down). We will assume that the system is in contact with a thermal reservoir at temperature T, in the canonical ensemble.

An important quantity in the Ising model is the magnetisation per spin:

$$M = \frac{1}{N} \sum_i^N \sigma_i, \tag{4}$$

which is simply the average value of the spin across the whole lattice. If $M = 0$, the system is unmagnetised, while if $M = \pm 1$, the system is completely magnetised (i.e. all spins align either up or down).

3.1 Statistical physics

Statistical physics predicts that the probability p_m of finding the system in a microstate m (corresponding to a particular configuration of the N spins) at temperature T is given by

$$p_m = \frac{1}{Z} e^{-E_m/k_B T}, \tag{5}$$

where E_m is the energy of the state m, and Z is the partition function,

$$Z = \sum_m e^{-E_m/k_B T}, \tag{6}$$

which sums up the statistical weighting of every state available to the system. Note that if we sum the probabilities for all possible states, we find that

$$\sum_m p_m = \frac{1}{Z} \sum_m e^{-E_m/k_B T} = \frac{Z}{Z} = 1, \tag{7}$$

i.e. the probability of finding the system in *some* state is 1.

To measure the statistical average of some observable O (e.g. the magnetisation or the energy), we can take an average of that observable over all possible states, weighted by the probability of finding the system in each state:

$$\langle O \rangle = \sum_m O_m p_m = \frac{1}{Z} \sum_m O_m e^{-E_m/k_B T}, \tag{8}$$

where O_m is the value of the observable measured for a particular state m.

3.2 Exercise – brute force calculation of spin statistics

We will start exploring the Ising model by performing a brute force calculation of all possible microstates, and the resulting statistics that we get from the expressions defined above. For N spins, there are 2^N possible configurations (because each spin has 2 possible states). Importantly, we need to know the details of *all* of these states if we want to use the definitions above to access information about the macroscopic behaviour of the system.

Some code, "bruteForce.cpp", has been provided as a base for you to work with. Ultimately, you want to calculate $\langle E \rangle$ and $\langle M \rangle$ (note that these are proportional to each other in the non-interacting Ising model). You could also put your spins into a vector or array, and then write some functions that calculate E and M with the array as an input. This might be slower in this example, but it's a good way to set up the code for the later workshops and the assignment.

The analytical result for the mean magnetisation and the mean energy in the non-interacting Ising model can be derived from considering the mean behaviour of a single spin, and the result is:

$$M = \tanh\left(\mu B / k_B T\right), \qquad E = -BMN \tag{9}$$

How do your results compare to this prediction? Set $\mu = B = k_B = 1$, and try a few temperatures.

The provided code also suggests that you construct and output the multiplicity function $g(E)$, i.e. the number of microstates with energy E. Once you've done this, plot it and see what the distribution looks like. What do you expect it to look like? How does the magnetisation of the most degenerate state compare to the mean magnetisation you measure? In what limit do the two converge?

3.3 Exercise – Markov chain Monte Carlo, done badly

In general, calculating thermodynamic quantities using the brute force method is tedious (as I'm sure you'll agree if you've done the previous exercise), and is not feasible for large configurations of spins. As such, we need a more sophisticated tool for calculating the thermodynamic behaviour of our system—this is where Monte Carlo methods come in. Here, we will set up a Markov chain Monte Carlo algorithm, but with a "bad" rule for flipping spins; namely, we will just flip them at random (we'll get to the "proper" algorithm in the next workshop).

Write a code that initiates a 1D array of spins (e.g. using the matrix class provided, or a `vector<int>`) that all start in the $\sigma = +1$ state. Start with $N \approx 10$. Then set up a loop that iterates a large number of times, and for each iteration, chooses a spin at random and flips it (this is your Markov chain).

Part A – Using the Markov chain as a random sample of all microstates:
If we generate enough states in the above manner, we will *eventually* sample all possible configurations, and we'll find that the microstates appear at the same rate as predicted by the multiplicity function *on average*. [As an analogy, think about rolling two dice—if we roll them enough times, eventually we will see that (for example) the number 7 comes up $\approx 1/6$th of the time, because there are 6 ways of obtaining this "macrostate" out of 36 total "microstates". But our measurements of these percentages will only converge in the limit of $\gg 36$ rolls.]

In your Markov chain loop, keep track of all the same observables as in the "brute force" exercise (this includes the partition function, so you'll need to set a temperature here too). What happens when you do this? What are your final $\langle E \rangle$ and $\langle M \rangle$ observables after normalising by the partition function? You can check the accuracy of the mean magnetisation for a given temperature by outputting e.g. $|\langle M \rangle - M_{\text{theory}}|/M_{\text{theory}}$ (see Eq. 9 for M_{theory}). Try a few different temperatures. You may notice that the accuracy becomes particularly bad at low temperatures—why is this? Think about how the random over-counting and under-counting of microstates affects your weightings. How will the accuracy of your multiplicity function vary as you change the number of spins and the number of samples in your Markov chain? How many samples will you need to achieve adequate sampling for, say, 100 spins?

Part B – Using the Markov chain as an importance sampled distribution of microstates:
If we were implementing importance sampling in this Markov chain (e.g. using the Metropolis algorithm), we would not need to worry about calculating Boltzmann factors and the partition function in our loop in order to estimate our expectation values $\langle E \rangle$ and $\langle M \rangle$. Rather, we would simply calculate E and M for each state we produce, and average over them at the end. In this case, the importance sampling scheme will take care of the weighting for us (because the Metropolis algorithm includes the Boltzmann factors). Importance sampling ensures that we don't need to generate a multiplicity distribution that adequately samples all possible microstates—we instead focus our sampling on the region of the energy spectrum that is relevant at the chosen temperature.

Run your code again, but this time, treat it as if importance sampling had been implemented. To do this, calculate E and M for each state sampled, and then average over these values at the end of the Markov chain. What temperature will our algorithm be sampling, given that we're just flipping spins at random?

Save the energy and magnetisation measurements to a file each iteration, and plot them as a function of the iteration number. If you look at the early times, you should see some evidence of "burn-in" (try increasing N to see this more clearly). You might also like to occasionally output the current spin configuration to the terminal, e.g. in a form like:

```
O - O O O - - O - O O - - O
```

where "O" corresponds to $\sigma = +1$, and "-" corresponds to $\sigma = -1$.