Computational Physics: Advanced Monte Carlo Methods

Thibeau Wouters

December 2020

Contents

1 Non-uniform random numbers

Central to any Monte Carlo method or Monte Carlo algorithm are random numbers. A first step towards creating such algorithms to solve problems in physics is understanding how exactly we can generate random numbers. In this section, we investigate how one can get random numbers from any continuous probability distribution f while only generating random numbers from the simplest possible distribution, namely the uniform distribution in the interval $[0, 1]$, also called the standard uniform distribution.

We present and investigate two methods to obtain non-uniform random numbers while only making use of Python's `random.uniform` function to generate random numbers.

1.1 Inverse transform random sampling

Proposition 1.1 (Probability integral transform)**.** *Suppose the random variable X satisfies $X \sim \mathcal{U}(0, 1)$, f is a continuous probability distribution with corresponding cumulative distribution F and $Y = F^{-1}(X)$. Then Y has the cumulative distribution function F.*

Proof. We will show that $P(Y \leq y) = F(y)$. For this, note that

$$\begin{aligned} P(Y \leq y) &= P(F^{-1}(X) \leq y) \\ &= P\left(F(F^{-1}(X)) \leq F(y)\right) \\ &= P(X \leq F(y)) = F(y) \end{aligned}$$

since F is by definition a non-decreasing function, and for all $x \in [0,1]$, we have $P(X \leq x) = x$, since $X \sim \mathcal{U}(0,1)$. □

We now illustrate this method by applying it to three different functions f.
First function: Consider the probability density function f_1 of a $\mathcal{U}(-2,1)$ distribution, i.e.:

$$f_1 : \mathbb{R} \to \mathbb{R} : x \mapsto \begin{cases} \frac{1}{3} & x \in [-2,1] \\ 0 & \text{elsewhere} \end{cases} . \tag{1.1}$$

Then the inverse of the cumulative distribution function F_1 is

$$F_1^{-1} : [0,1] \to [-2,1] : x \mapsto \frac{x+2}{3} . \tag{1.2}$$

Second function: Consider the probability density function f_2 of an exponentially distributed random variable:

$$f_2 : \mathbb{R} \to \mathbb{R}^+ : x \mapsto \begin{cases} e^{-x} & x \geq 0 \\ 0 & x < 0 \end{cases} . \tag{1.3}$$

The inverse of the cumulative distribution function F_2 is given by

$$F_2^{-1} : [0,1] \to \mathbb{R}^+ : x \mapsto -\log(1-x) . \tag{1.4}$$

Third function: Consider the linear probability function

$$f_3 : \mathbb{R} \to \mathbb{R} : x \mapsto \begin{cases} \frac{x+1}{2} & x \in [-1,1] \\ 0 & \text{elsewhere} \end{cases} . \tag{1.5}$$

The inverse of the cumulative distribution function F_3 is given by

$$F_3^{-1} : [0,1] \to [-1,1] : x \mapsto 2\sqrt{x} - 1 . \tag{1.6}$$

In Figure X below, we plot the histograms for 10^5 generated random numbers that we obtain for these three distributions, along with the expected density functions. The histograms clearly match the desired distributions, and hence the probability integral transformation is a useful random number generator.

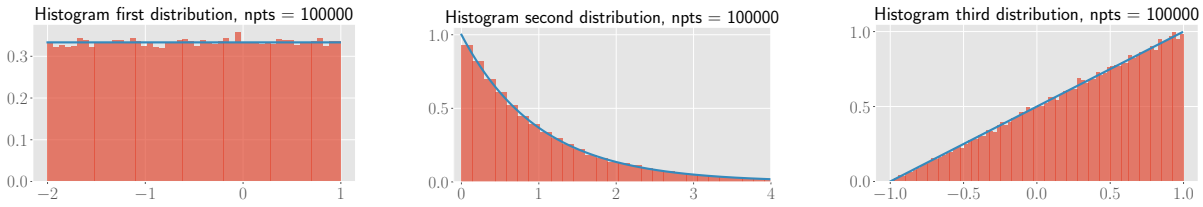

Figure 1.1: Histograms of 10^5 generated random numbers for the three distributions, along with the expected density function shown in blue.

1.2 Hit-and-miss method

The hit-and-miss method works for distributions f that are only non-zero in a finite interval $[a, b]$. Let M be a number that is greater than $f(x)$ for all $x \in [a, b]$. We generate t and s out of a $\mathcal{U}(a, b)$ and a $\mathcal{U}(0, M)$ distribution, respectively. If $s \leq f(t)$, then this is called a 'hit' and the value t is stored. If $s > f(t)$, then this is called a 'miss', and the value is not returned. It is immediately clear that the hit-and-miss method therefore wastes computer resources whenever a miss is generated, which is a drawback of the algorithm. Note that the hit-and-miss method cannot be used for the exponential distribution (defined as f_2 above), since this distribution is non-zero on an infinite interval.

We illustrate the hit-and-miss method for two distributions: the first is the function f_3 defined above. The histogram that we obtain is very similar to the one on the right in Figure (1.1) (see the Jupyter notebook).
We can quickly compare the two methods for generating random numbers described in this section for this particular distribution. In the notebook, a command cell is written which generates 10^6 random numbers using these two algorithms and computes the time that is necessary for this computation. This is repeated 10 times, and we compare the averages. While the two average times do not differ by a lot, the hit and miss method is slightly slower. The first method requires on average 0.7466 seconds of computation time, while for the hit and miss method, this is 0.9862 seconds. However, the hit-and-miss method only keeps on average 45.4688 % of the random numbers it generated, so most of the computation time is wasted. This example shows that whenever the cumulative distribution function F can easily be inverted, as was the case for the functions in the previous subsection, then the probability integral transformation formula is the preferred method for generating random numbers.

However, the hit-and-miss method does have some advantages over the probability integral transformation. Consider for example the function f_4 defined by

$$f_4 : [0, 1] \to -x(1 - x)e^{-x^2 \log(1-x)} . \tag{1.7}$$

Even though the hit-and-miss method wastes computer resources, as mentioned above, it can be preferred over the probability integral transformation for more complicated functions such as f_4, since the user does not have to solve and invert an integral equation. Indeed, the probability integral transformation formula cannot easily be applied to those complicated functions, and one example of such a function is the normal distribution, which is very common in both mathematics and physics. For these functions, the hit-and-miss method is still valid and easy to implement. We show a histogram in Figure 1.2 containing 10^5 points of the distribution with probability density proportional to f_4. Note that f_4 is not normalised: the normalisation constant N is calculated using Python. We denote the normalised density function as $f_{4,N} = Nf_4$

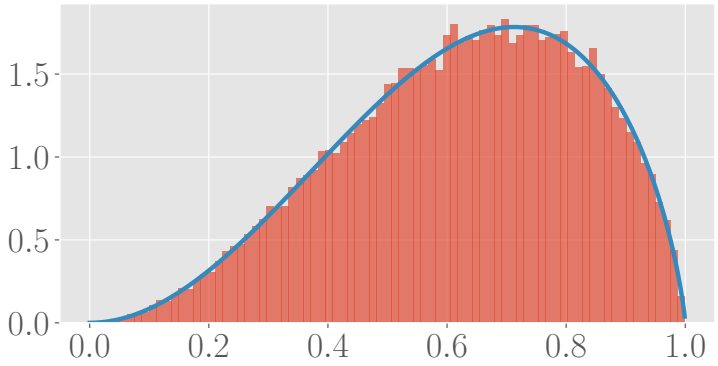

Figure 1.2: Histogram of 10^5 generated random numbers using the hit-and-miss method, along with the normalised density function $f_{4,N}$ shown in blue.

1.3 Uniform random points in a circle

The goal is to generate random points uniformly in a circle of radius R and centre (x_0, y_0) in two ways: the first one makes use of the hit-and-miss method, while the second one uses an appropriate distribution function $f(r, \theta)$, where r, θ are polar coordinates.

The idea behind the hit-and-miss algorithm is as follows. Given a radius R, we generate two random numbers x and y from a $\mathcal{U}(-R, R)$ distribution. If the distance between the point (x, y) and the centre (x_0, y_0) is greater than R, this is a 'miss' and the function does not return anything. If this distance is at most R, the point is saved. We let the hit-and-miss algorithm run 10^5 times and plot the 2D histogram of generated points in Figure 1.3: the points are indeed uniformly distributed in the circle.

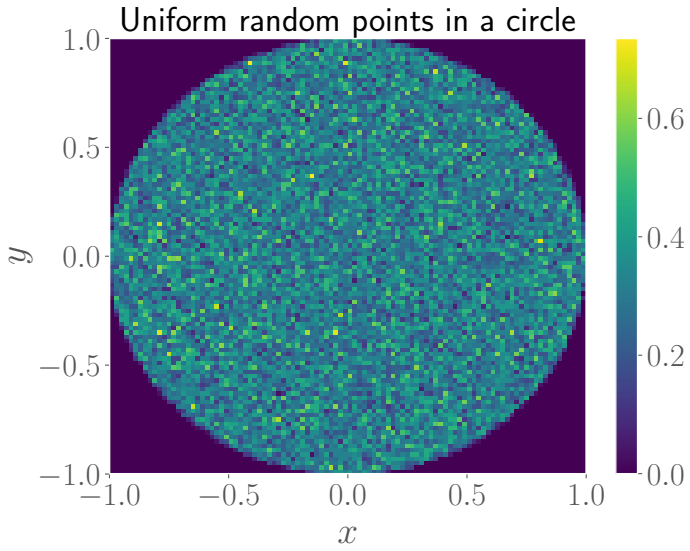

Figure 1.3: 10^5 points uniformly distributed in the unit circle, generated using the hit-and-miss algorithm.

Now we generate the random points using an appropriate distribution function $f(r, \theta)$.

We can in fact separate the variables for this function, and look for appropriate distribution functions $\mathcal{R}(r), \mathcal{T}(\theta)$ to generate the random points. For $\mathcal{T}$, we use a uniform distribution in the interval $[0, 2\pi]$. One could naively think that we could also generate r out of a uniform distribution on $[0, R]$, but this is in fact not true. Indeed, Figure 1.4 shows the result in that case: these points are clearly not uniformly distributed: we generate more points closer towards the origin.

To understand the problem, we note that if we look at a certain interval for the angle θ, say $[\theta, \theta + \mathrm{d}\theta]$, then the length of the arc between (r, θ) and $(r, \theta + \mathrm{d}\theta)$ is equal to $L = r\,\mathrm{d}\theta$. Hence for the same range of the angle, more points should be generated further away from the origin, in a linear way.

This thought experiment indicates we should generate the radius r from the distribution

$$\mathcal{R} : [0, R] \to \mathbb{R} : r \mapsto \frac{2}{R^2} r\,, \tag{1.8}$$

where the factor $2/R^2$ is used for normalisation. By applying Proposition 1.1, this can be done by generating a random number from the standard uniform distribution, and transforming it via the function

$$\mathcal{R}^{-1} : [0, 1] \to [0, R] : x \mapsto \mathcal{R}^{-1}(x) = R\sqrt{x}\,. \tag{1.9}$$

This gives the correct distribution function for the radial coordinate. Figure 1.4 shows the points generated by $f(r, \theta) = \mathcal{R}\mathcal{T}(\theta)$. This is indeed the correct distribution to generate points uniformly in a circle.

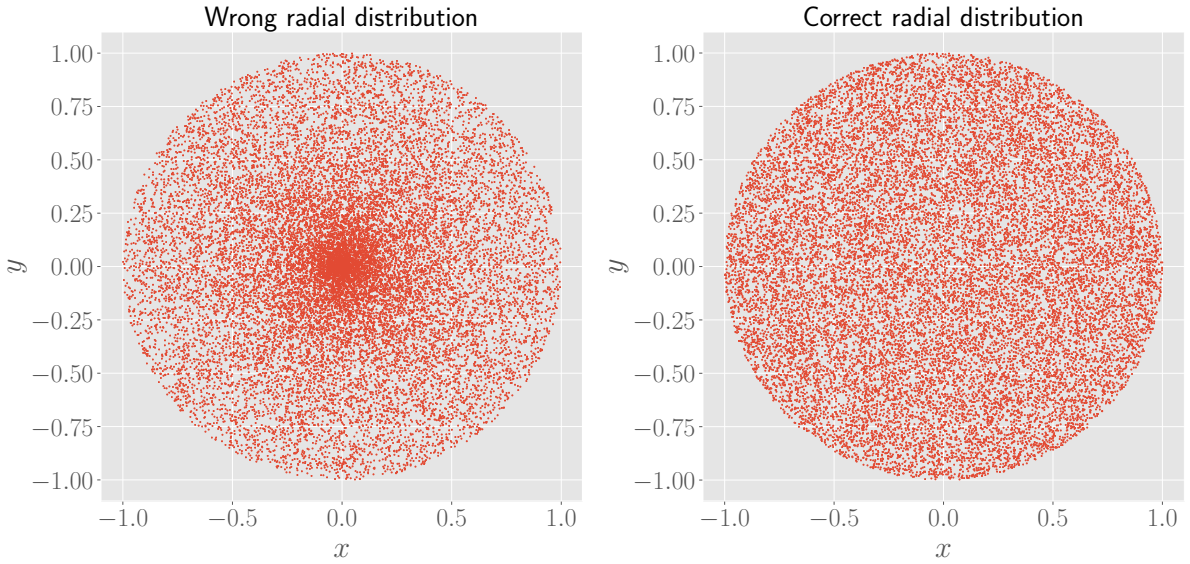

Figure 1.4: Left: Illustration that a uniform distribution for the radial coordinate will not generate points uniformly inside a circle. *Right:* An illustration of the correct radial distribution (see text). In both figures, $2 \cdot 10^4$ points are shown.

2 Gaussian RNG

The method in the previous section of the probability integral transformation cannot be applied to one of the most common distributions in all of physics: the Gaussian distribution. Therefore, we try to look for another way to generate random numbers from a Gaussian distribution in this section, which is known as the Box-Muller transformation.

If we draw x and y from a standard Gaussian distribution, the joint density function is

$$f(x,y)\,\mathrm{d}x\,\mathrm{d}y = \frac{1}{2\pi}\exp\left(-\frac{x^2+y^2}{2}\right)\mathrm{d}x\,\mathrm{d}y\,, \tag{2.1}$$

which can be transformed into polar coordinates:

$$f(r,\theta)\,\mathrm{d}r\,\mathrm{d}\theta = \frac{1}{2\pi}\exp\left(-\frac{r^2}{2}\right) r\,\mathrm{d}r\,\mathrm{d}\theta\,. \tag{2.2}$$

So the idea behind the Box-Muller transformation is to generate two random numbers r, θ from the distribution (2.2), such that $x = r\cos\theta$, $y = r\sin\theta$ are two random numbers from a standard Gaussian. The number θ is drawn from a $\mathcal{U}(0, 2\pi)$ distribution, while r can be obtained via a probability integral transformation. For this, we have to invert

$$F(r) = \int_0^r \exp\left(-\frac{z^2}{2}\right) z\,\mathrm{d}z\,, \tag{2.3}$$

which, contrary to the density function of a standard Gaussian in one dimension, can more easily be inverted. The trick behind Box-Muller is essentially to go to two dimensions, and benefit from the Jacobian of polar coordinates: this gives a factor r, which makes that we can solve the integral above. Indeed, if we make the change of variables $u = z^2/2$, then $z\,\mathrm{d}z = \mathrm{d}u$ and the integral evaluates to

$$F(r) = 1 - \exp\left(-\frac{r^2}{2}\right)\,. \tag{2.4}$$

The inverse function is therefore

$$F^{-1} : [0,1] \to \mathbb{R}^+ : x \mapsto \sqrt{-2\log(1-x)}\,, \tag{2.5}$$

and we are now able to generate random numbers r from the distribution in (2.2).

We write a Python function that generates one couple of random numbers x and y from a standard Gaussian distribution using the algorithm described above. The parameter npts represents the number of times we repeat this algorithm. In Figure 2.1, we show the histograms that we obtain after npts $= 10^5$ repetitions of the function. The joint probability distribution indeed resembles the density plot of a two-dimensional Gaussian distribution, and the one-dimensional histograms clearly follow a Gaussian distribution, of which the graph is shown in blue.

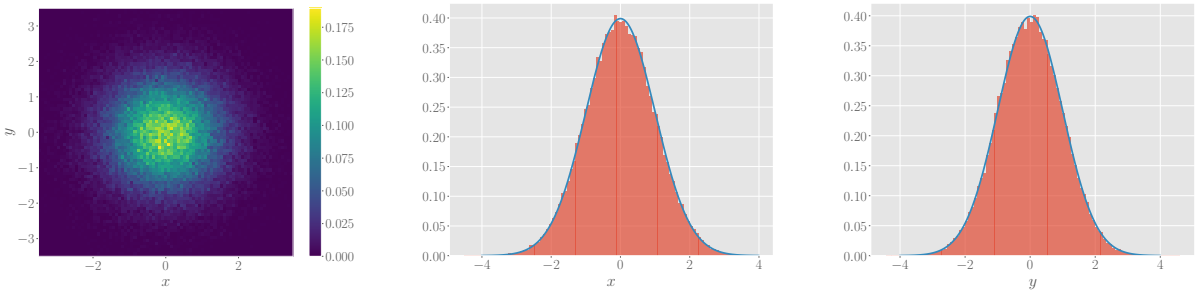

Figure 2.1: From left to right: a two-dimensional density plot of (x, y) pairs, and histograms for x and y after applying the algorithm npts $= 10^5$ times. The blue curve on top of the 1D histograms is a plot of the standard Gaussian distribution.

As we saw, we are now able to generate random numbers from a Gaussian distribution with $\mu = 0$ and $\sigma = 1$. However, the above algorithm can easily be extended such that random numbers from *any* normal distribution $\mathcal{N}(\mu, \sigma)$ can be generated. This is done via a change of variables: if $Z \sim \mathcal{N}(0, 1)$, then the variable $X = \sigma Z + \mu$ is normal distributed with mean μ and standard deviation σ. This is implemented in Python, and we illustrate this for $\mu = 2$ and $\sigma = \frac{1}{2}$ in Figure 2.2, which is similar to Figure 2.1 for the standard Gaussians. Also shown are the probability distributions of $\mathcal{N}(\mu, \sigma)$ on top of the 1D histograms.

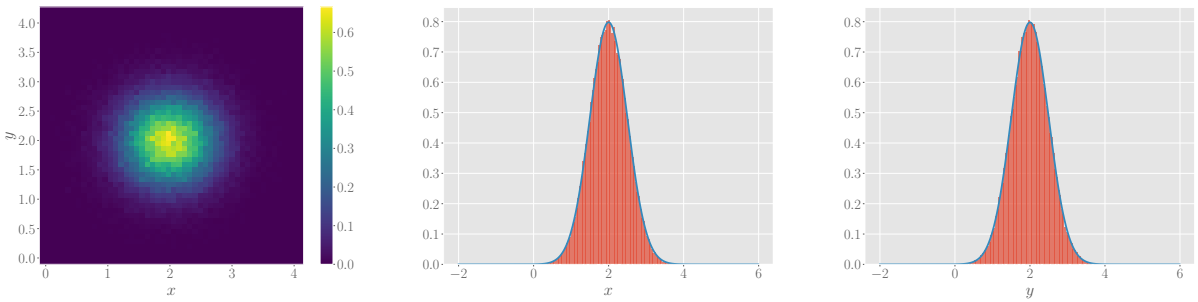

Figure 2.2: From left to right: a two-dimensional density plot of (x, y) pairs, and histograms for x and y after applying the algorithm npts $= 10^5$ times with $\mu = 2$ and $\sigma = \frac{1}{2}$. The blue curve on top of the 1D histograms is a plot of the Gaussian distribution $\mathcal{N}(\mu, \sigma)$.

3 Monte Carlo integration (I)

Now that we understand the methods behind generating random numbers from continuous distributions, we will go to a first and one of the most common applications of the Monte Carlo method, namely numerical approximation of integrals. We will estimate the following integral by relying on the Monte Carlo methods seen in the course notes:

$$\mathcal{I} = \int_{\mathbb{R}^3} \exp\left(-\frac{r^2}{2}\right) (x + y + z)^2 \, \mathrm{d}x \, \mathrm{d}y \, \mathrm{d}z \,, \tag{3.1}$$

where $r^2 = x^2 + y^2 + z^2$. We first compute this result analytically, before approximating it using the Monte Carlo method.

3.1 Analytical calculation

We expand the square:

$$\mathcal{I} = \int_{\mathbb{R}^3} \exp\left(-\frac{r^2}{2}\right)(x^2+y^2+z^2+2xy+2yz+2xz)\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z\,. \tag{3.2}$$

We see that there are only two different sorts of integrandi. Indeed, it is sufficient to calculate the following two integrals analytically:

$$I_1 = \int_{\mathbb{R}^3} x^2 \exp\left(\frac{x^2+y^2+z^2}{2}\right)\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z, \quad I_2 = 2\int_{\mathbb{R}^3} xy \exp\left(\frac{x^2+y^2+z^2}{2}\right)\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z\,, \tag{3.3}$$

since then the integral that we wish to compute is simply $\mathcal{I} = 3I_1 + 3I_2$. The integral I_2 is equal to zero: we can factorise the exponential, and then note that we need to compute an integral over x of an odd function in x with domain $\mathbb{R}$, which necessarily vanishes. The value of I_1 can be obtained from the well-known Gaussian integral

$$\mathcal{G}_\alpha = \int_{\mathbb{R}} \exp\left(-\alpha x^2\right)\mathrm{d}x = \sqrt{\frac{\pi}{\alpha}}\,, \tag{3.4}$$

and, by viewing it as a function of α, we also obtain that

$$\int x^2 \exp\left(-\alpha x^2\right)\mathrm{d}x = -\frac{\mathrm{d}}{\mathrm{d}\alpha}\mathcal{G}_\alpha = \frac{\sqrt{\pi}}{2}\alpha^{-3/2}\,. \tag{3.5}$$

All together, we find the value of I_1 to be

$$\begin{aligned} I_1 &= \int_{\mathbb{R}} x^2 \exp\left(-\frac{x^2}{2}\right)\mathrm{d}x \int_{\mathbb{R}} \exp\left(-\frac{y^2}{2}\right)\mathrm{d}y \int_R \exp\left(-\frac{z^2}{2}\right)\mathrm{d}z \\ &= \left(\mathcal{G}_{1/2}\right)^2 \left(-\frac{\mathrm{d}\mathcal{G}_\alpha}{\mathrm{d}\alpha}\right)\Big|_{\alpha=\frac{1}{2}} = (2\pi)^{3/2}\,. \end{aligned}$$

So we conclude that

$$\mathcal{I} = 3(2\pi)^{3/2}\,. \tag{3.6}$$

3.2 Numerical calculation using the Monte Carlo method

We now approximate the integral via the Monte Carlo methods discussed in the lecture notes, using the factor $\exp\left(-\frac{r^2}{2}\right)$ as a density function. We have first to impose a normalisation to this function, i.e. we have to choose N such that the function $W(x,y,z) = N\exp\left(-\frac{x^2+y^2+z^2}{2}\right)$ is normalised to one. We calculate, using the results mentioned above, that

$$N\int_R \exp\left(-\frac{x^2+y^2+z^2}{2}\right)\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z = N\left(\int_{\mathbb{R}} \exp\left(-\frac{x^2}{2}\right)\mathrm{d}x\right)^3 = N(2\pi)^{3/2}\,, \tag{3.7}$$

and hence the normalised density function is

$$W(x,y,z) = \frac{1}{(2\pi)^{3/2}}\exp\left(-\frac{x^2+y^2+z^2}{2}\right)\,, \tag{3.8}$$

which is the density function of a standard multivariate normal distribution in three variables.

To approximate the integral, let f denote the integrand, i.e. $\mathcal{I} = \int_{\mathbb{R}^3} f(x,y,z)\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z$ and let $g(x,y,z) := (x+y+z)^2$. Since f can be factorised in a function proportional to W multiplied with the function g, we have that

$$\mathcal{I} = \int_{\mathbb{R}^3} f(\mathbf{x})\,\mathrm{d}^3\mathbf{x} \approx \frac{1}{N}\sum_{n=1}^{N}\frac{f(\mathbf{x}_n)}{W(\mathbf{x}_n)} \approx \frac{1}{N}(2\pi)^{3/2}\sum_{n=1}^{N} g(\mathbf{x}_n)\,, \tag{3.9}$$

where $\mathbf{x} \equiv (x,y,z)$ for notational simplicity. The points $\mathbf{x}_n, n = 1,\ldots,N$ are generated according to the distribution $W(\mathbf{x}_n)$. Comparing this with our result for $\mathcal{I}$ in equation (3.6), we see that in order to have a good approximation, we would like $\frac{1}{N}\sum_{n=1}^{N} g(\mathbf{x}_n) \approx 3$.

We now implement the above algorithm to compute the integral numerically. We choose $N = 10^4$ (this is called npts in the Python notebook) and repeat this calculation 100 times. In Figure 3.1, we show a plot of the obtained approximations along with the exact value that we computed analytically.

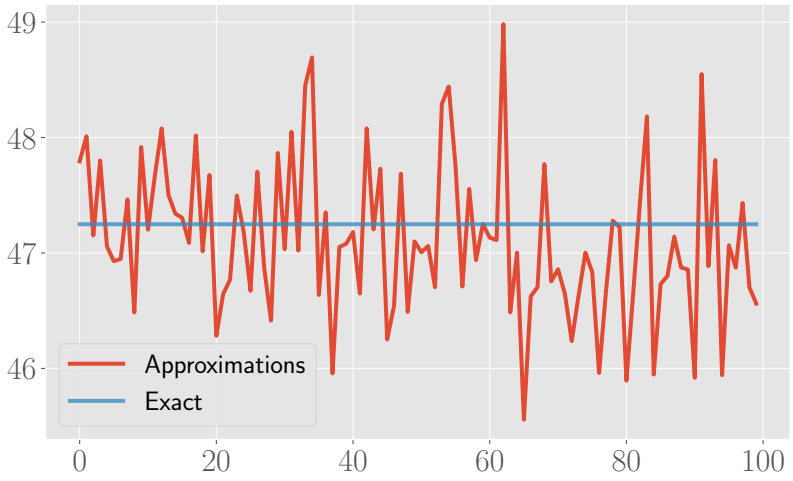

Figure 3.1: Plot of 100 approximations of $\mathcal{I}$ with $N = 10^4$ for each approximation, along with the exact value.

In one of the runs, the average of our 100 approximations of the integral was equal to 47.1200, while the exact value is 47.2488 (rounded to four decimals). The sample standard deviation was 0.6609, such that the exact value lies within the range of our error. Hence the approximation is reasonably good.

We now look at the variances: we are interested in the behaviour of σ^2 of the sampled values for different numbers of sampled points N. We let the algorithm run with N ranging from 50 to 6000, calculating σ^2 for each value of N. Figure 3.2 below shows the resulting values of σ^2. We can recognise the behaviour $\sigma^2 \sim 1/N$ in the sampled data points. In order to firmly establish this relation, we fit a linear curve through the points $(N, 1/\sigma^2)$ with the Scipy package. The result is

$$1/\sigma^2 = m \cdot N + c, \quad m = 0.0105; c = 0.5559, \quad R^2 = 0.93221\,. \tag{3.10}$$

Figure 3.2 also contains a plot showing the $1/\sigma^2$ data points along with the linear fit. Since the coefficient of determination is fairly close to one for this linear fit, we can indeed conclude that $\sigma^2 \sim 1/N$.

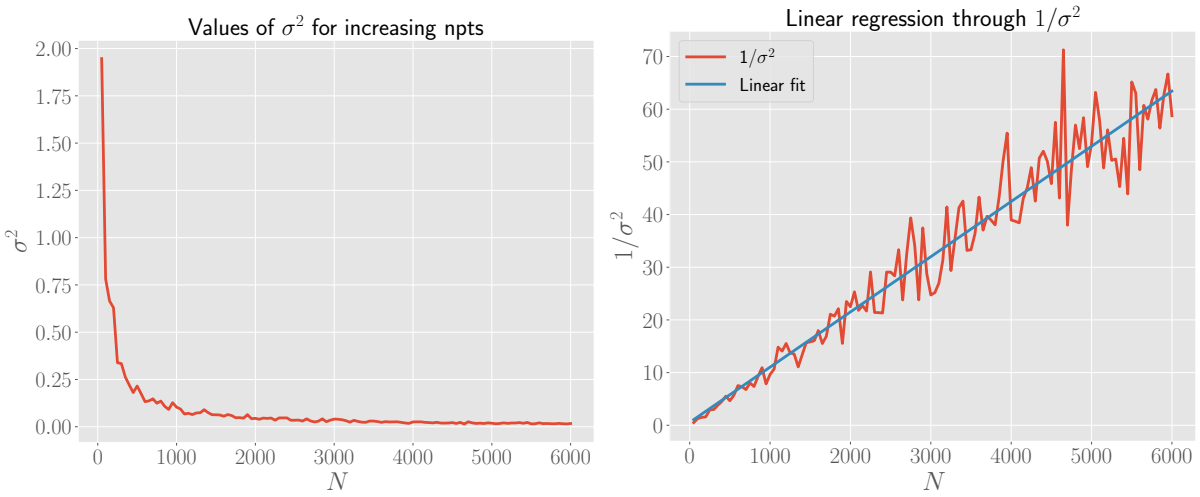

Figure 3.2: Left: Plot of the variances σ^2 for increasing values of N. *Right*: Linear regression through the points $(N, 1/\sigma^2)$. Parameters of the fit are given in the text.

As a final remark, we note that we cannot change the roles of the factors of the integrand. That is, the factor $(x+y+z)^2$ cannot be used as density, and $\exp\left(-\frac{r^2}{2}\right)$ for the function g as defined above. This is because the factor $(x+y+z)^2$ cannot be normalised in the domain of integration, which we have to require of a density function. Indeed, it is clear that we cannot choose a $N \in \mathbb{R}$ such that

$$N \int_{\mathbb{R}^3} (x+y+z)^2 \, \mathrm{d}x \, \mathrm{d}y \, \mathrm{d}z = 1 \,, \tag{3.11}$$

and hence this factor cannot be used to define a density function.

4 Monte Carlo integration (II)

We again try to numerically approximate the integral from the previous section

$$\mathcal{I} = \int_{\mathbb{R}^3} \exp\left(-\frac{r^2}{2}\right) (x+y+z)^2 \, \mathrm{d}x \, \mathrm{d}y \, \mathrm{d}z \,, \tag{4.1}$$

but now we define a Markov chain, characterised by the transition probabilities $P(\mathbf{r} \to \mathbf{r}')$, of which the dynamics converge to the distribution $w(\mathbf{r}) = \frac{1}{(2\pi)^{3/2}} \exp\left(-r^2/2\right)$. This will be the case, as seen in the lecture notes, if we require detailed balance and is achieved in practice by factorising the transition probabilities into a selection probability g and acceptance ratio A:

$$P(\mathbf{r} \to \mathbf{r}') = g(\mathbf{r} \to \mathbf{r}') A(\mathbf{r} \to \mathbf{r}') \,. \tag{4.2}$$

In our case, g is generated by going from $\mathbf{r}$ to $\mathbf{r}' = \mathbf{r} + \boldsymbol{\delta}$, where $\boldsymbol{\delta} = (\delta_x, \delta_y, \delta_z)$ and the components δ_i are chosen independently from a uniform distribution centered at the origin and of width h, called $W_h(\delta)$. The acceptance ratio is given by

$$A(\delta) = \begin{cases} 1 & \text{if } \; r'^2 < r^2, \\ \exp\left(-\frac{r'^2 - r^2}{2}\right) & \text{otherwise}\,. \end{cases} \tag{4.3}$$

In words, we certainly accept the transition if it brings us closer to the origin. However, if the transition brings us farther away from the origin, we accept it with a certain probability, and we 'flip a coin' to determine whether or not the move is accepted.

This algorithm is called the Metropolis algorithm. First, we investigate the behaviour and the equilibrium distribution of the random walk (RW) which is used in the Metropolis algorithm for various starting positions and values of h, which in essence determines the maximum size of a single step in the walk. The Metropolis algorithm is used to approximate the integral $\mathcal{I}$, and we compare the outcome with that of the Gaussian RNG from the previous section.

4.1 Trajectories of the Metropolis algorithm

Using the Metropolis algorithm, we plot the trajectory in the (x, y) plane of a walk consisting of $N = 10^4$ steps that the algorithm generates, starting from $\mathbf{r}_0 = (0, 0, 0)$ or $\mathbf{r}_0 = (10, 10, 10)$, and for $h = 1$ and $h = 0.1$. We will refer to this as a 'particle' which makes the RW.

We first plot the trajectory starting from the origin and both values of h in Figure 4.1. As expected, the particle stays in the neighbourhood of the origin, because the acceptance ratio tends to attract the particle towards the origin. Moreover, it stays closer to the origin for the smaller value of the width $h = 0.1$, while it explores more of the neighbourhood of the origin $h = 1$.

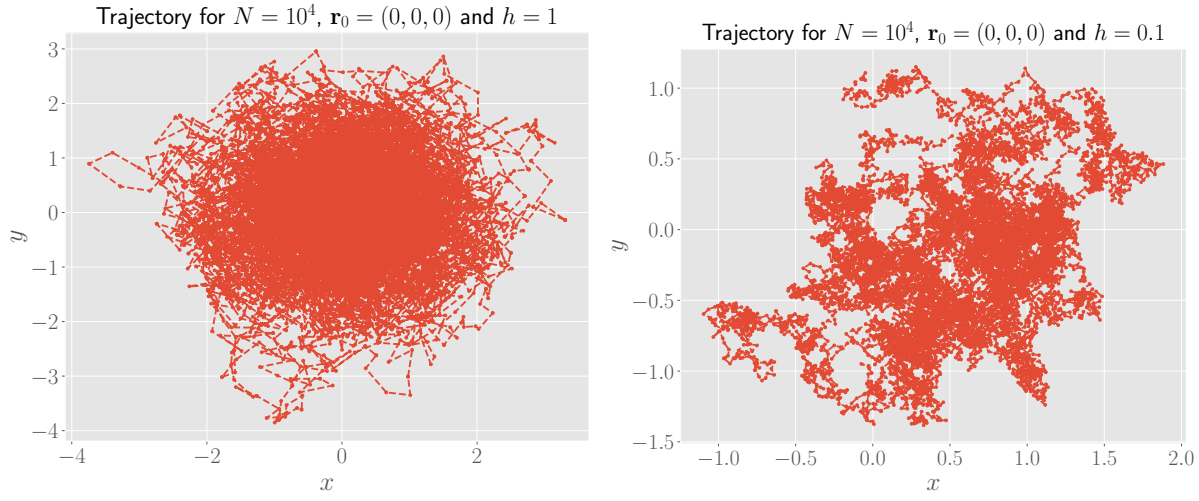

Figure 4.1: Two trajectories in the (x, y) plane of points generated with the Metropolis algorithm, starting from the origin. *Left:* $h = 1$, *Right:* $h = 0.1$.

We now repeat the above set-up for trajectories that start in the initial position $\mathbf{r}_0 = (10, 10, 10)$. The result is shown in Figure 4.2. As expected, the particle tends to move towards the origin in both cases. Since the width h determines how large the steps can be, this convergence towards the origin is faster for $h = 1$ compared to $h = 0.1$. Therefore, for a larger h, the particle has more time to explore the neighbourhood of the origin: indeed, after a few steps, the plot on the left hand side in Figure 4.2 resembles the plot where the trajectory started from the origin, see Figure 4.1. For a small step size, such as $h = 0.1$, this is not true: it takes almost the whole trajectory to come close towards the origin. This is studied in more detail in section 4.3.

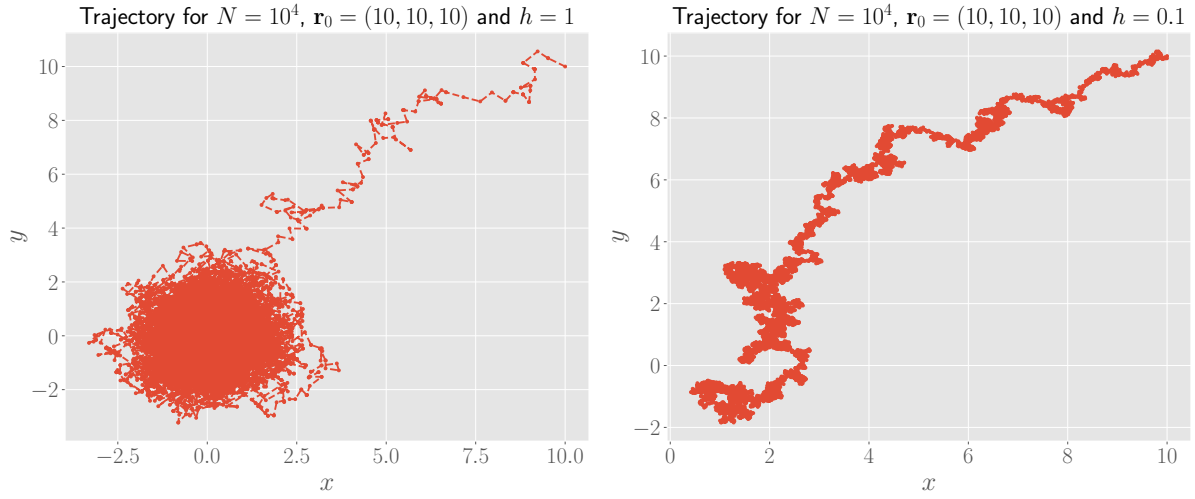

Figure 4.2: Two trajectories in the (x, y) plane of points generated with the Metropolis algorithm, starting from $(10, 10, 10)$. *Left*: $h = 1$, *Right:* $h = 0.1$.

If we perform the random walks $N = 10^6$ times, then the equilibrium distribution is a multivariate Gaussian distribution for three variables, centered at the origin. This can be verified by plotting the histograms for x, y and z, and we also show a 2D histogram in the (x, y)-plane. The result is shown in Figure 4.3.

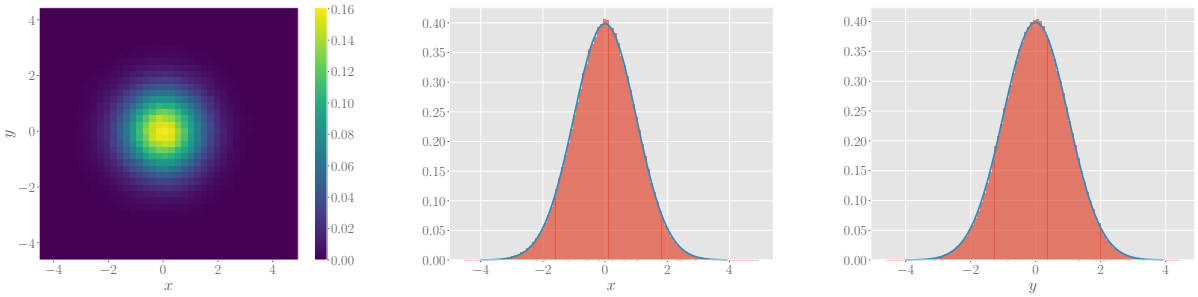

Figure 4.3: From left to right: A histogram of positions of a random walk ($N = 10^6$ steps) in the (x, y)-plane, and histograms for the x and y positions, along with a standard Gaussian distribution shown in blue.

4.2 Numerical approximation of the integral

We now apply the Metropolis algorithm to estimate the integral $\mathcal{I}$ given in equation (4.1). We perform a random walk, as defined above, starting from the origin. At each point (x, y, z) in the walk, we compute $g(x, y, z)$ and store this value. In the end, we return the average of these values. Recall from see equation (3.9) that we would like this to approximate 3, since multiplying this with the correct normalisation factor yields the integral $\mathcal{I}$.

In one of the runs, we let the algorithm compute the values of g at 10^5 points to approximate $\mathcal{I}$, and did this for 100 repetitions. The mean of the approximations was 47.4917, the exact value is 47.2488 (rounded to four decimals), and the sample standard deviation was 1.0954. Hence the approximation is reasonably good.

4.3 Equilibration

Starting from the initial point $\mathbf{r}_0 = (10, 10, 10)$, we estimate N_{eq}, the number of Monte Carlo steps necessary to reach the equilibrium distribution, for $h = 1$ and $h = 0.1$. We do this by making random walks with these configurations, and compute the distance r from the origin at each step. The results are given in Figure 4.4. For $h = 1$, equilibrium is reached around 200 Monte Carlo steps. For $h = 0.1$, this is around 6000 steps. As expected, the equilibrium is reached faster for a larger h, since h essentially represents the maximum size of each step in the random walk. Hence a bigger step size means the walk will converge towards the origin faster.

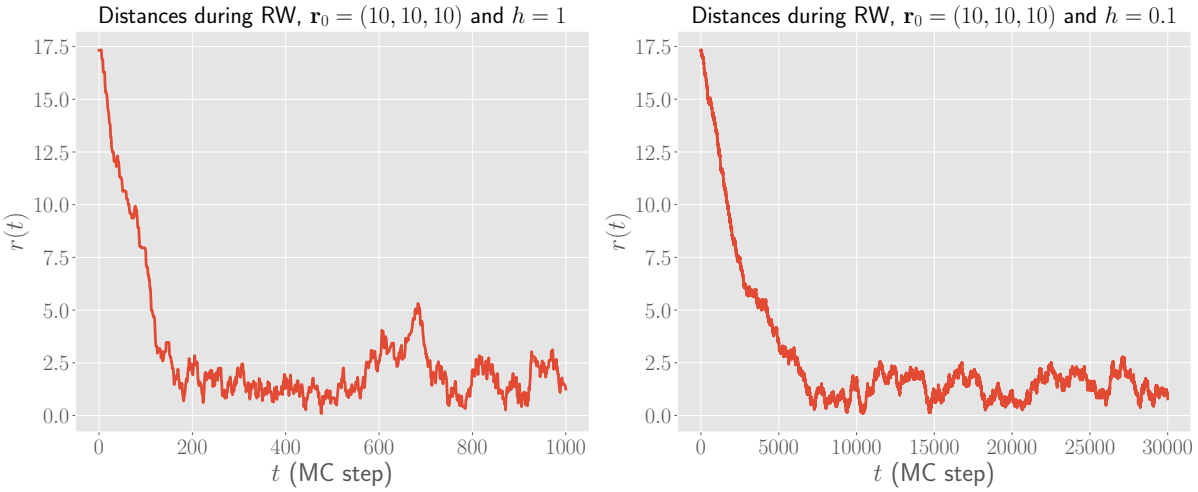

Figure 4.4: Plots of the distance from the origin at each step in a random walk, starting from $(10, 10, 10)$.

4.4 Comparison between Gaussian RNG with Metropolis

In the previous section, we studied the use of Gaussian RNG to approximate the integral $\mathcal{I}$. Now that we concluded our study of the Metropolis algorithm, we are able to com-

pare both methods. Both approximations were close to the exact value, and the sample standard deviations were in both cases large enough to explain the difference between the approximation and the exact value. So simply comparing the final outcomes of the algorithms does not give us a lot of insight.

As explained before, both algorithms sample points (x, y, z) in $\mathbb{R}^3$ in two different ways, and use these to compute evaluations of the function g, where $g(x, y, z) = (x + y + z)^2$, of which the average should approximate to three, and this approximation is used in equation (3.9). To compare the two methods, we can evaluate this approximation of three in both cases.

For this, we investigate the average of those function evaluations *at each step* of the approximation for both algorithms. We can then plot the average at each step. It is difficult to make a quantified comparison, and hence we take a qualitative approach and focus on aspects such as

- How quickly does the algorithm converge towards the exact value?
- Once equilibrium is reached, are the oscillations around the exact value large or small?
- Is the output of the algorithm consistent? Or do we see different results for different runs?

Before delving into the comparison of the two methods as described above, we need to realise that if we use the built-in functions of Python (e.g. `np.mean`) to compute the averages, this will be extremely inefficient for a large number of function evaluations, as we will now argue in more detail.

Suppose we did n function evaluations of g for our approximation and Python computed the average of these values. In the next step, we make a new evaluation of g, and add this to the list of all function evaluations. The built in function of Python then has to sum over all $n + 1$ values in the list, and divide by the length of the list. However, in the previous step, Python already summed over the first n values of the list, but this information is not used and we waste computer resources. To solve this issue, and make our algorithm much more efficient, we present a simple formula to compute by induction the average of a list of values to which a single value gets added.

Assume we have a list of length $n \geq 1$ of values $\{x_1, \ldots, x_n\}$ and know the average of this list:

$$\mu_n = \frac{1}{n} \sum_{i=1}^{n} x_i \,. \tag{4.4}$$

Suppose we add the value x_{n+1} to this list and want to know the average of all $n + 1$ values. Then this average is given by

$$\mu_{n+1} = \frac{n}{n+1} \mu_n + \frac{x_{n+1}}{n+1} \,. \tag{4.5}$$

It is immediately clear that this method can be applied to our approximations of the integral, and is much faster than using the built-in functions of Python. Indeed, the

built-in functions would need to make a number of computations which increases linearly with the length of the list, while the above method requires a fixed number of operations, independent of the length of the list.

This idea is implemented in a function which returns the average of the function evaluations of g at each Monte Carlo step for both algorithms. We are now ready to compare the Gaussian RNG with the Metropolis algorithm. In Figure 4.5, we make $2 \cdot 10^5$ Monte Carlo steps for both algorithms, compute $g(x, y, z)$ at each step, and plot the average of the function evaluations of g at each step. Since the averages vary wildly for the first few Monte Carlo steps, we omit the first 5000 computed averages from the plot. For the Metropolis algorithm,

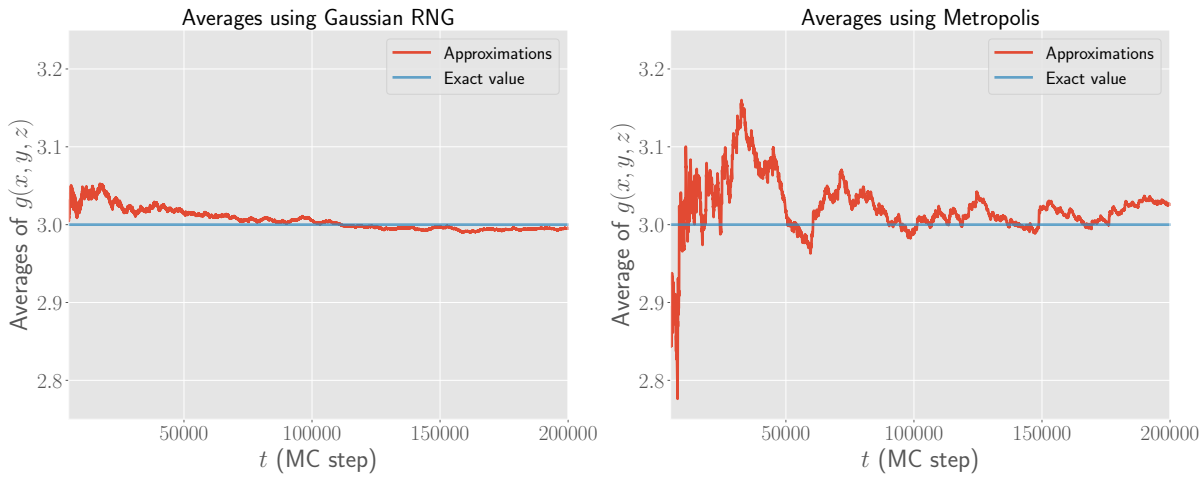

Figure 4.5: Comparison between the Gaussian RNG algorithm and the Metropolis algorithm to approximate the value three. Both plots start at $t = 5000$ MC steps. For the Metropolis algorithm, we used a random walk starting from the origin and $h = 1$.

Referring to our list of bullet points about comparing the two methods, we see that the Gaussian RNG method is preferred over the Metropolis algorithm. There are other disadvantages of the Metropolis algorithm. First of all, the output of the Metropolis algorithm is inconsistent. That is, in some runs, the approximation starts off with a large fluctuation, and decays away rather slowly, causing the approximation to converge slowly to the exact value such that after a large number of steps, it does not yield a good approximation. Moreover, this run started from the origin, which is around the equilibrium of the random walk, and with a width $h = 1$ in order to sample evenly distributed around the origin. Suppose we would start from $\mathbf{r}_0 = (10, 10, 10)$ and $h = 0.1$, the random walk has to equilibrate around 6000 steps (see previous subsection) before we can start computing function evaluations of g, otherwise this would cause a large fluctuation in the approximation. So in essence, we need to understand the dynamics generated by the Metropolis algorithm very well in order to get a random walk which is optimal for our approximation, and even then, the results can vary wildly, as seen in the Figure above. Hence we can conclude that the Gaussian RNG is much more simpler, both in theory as in implementation, and also more reliable for approximating the integral.

5 Stochastic Matrices

In this problem, we will delve deeper in the mathematics behind stochastic matrices, which appear all over statistical physics and in various Monte Carlo methods.

Meer intro?

Proposition 5.1. *Suppose we are given a non-negative initial vector w. Then the stochastic matrix generates non-negative vectors at all later times.*

Proof. The stochastic matrix lets the system evolve according to the prescription that if the system is at time t described by $w(t)$, then at a later time step $t + \Delta t$, it is described by the vector

$$w(t + \Delta t) = Pw(t)\,, \tag{5.1}$$

where

$$P : (P)_{\mu\nu} = P(\nu \to \mu) \tag{5.2}$$

is the stochastic matrix. By definition, all the entries of the stochastic matrix are non-negative. Therefore, if we assume that the vector $w(t)$ has only non-negative entries and we expand the matrix product in equation (5.1), we end up with a sum of products of non-negative quantities, hence these are again non-negative. So we conclude that $w(t + \Delta t)$ is again a non-negative vector.

This immediately implies by an inductive argument that if we start with an initial non-negative vector w, the stochastic matrix generates non-negative vectors at all later times. □

Proposition 5.2. *Assume that P and Q are two $N \times N$ stochastic matrices. Then PQ is a $N \times N$ stochastic matrix.*

Proof. To prove this, we have to show that

i. all the elements of PQ are non-negative,

ii. that the sum of the columns of PQ are equal one.

To prove i., we note that the an element of the matrix PQ is simply

$$(PQ)_{ij} = \sum_{k=1}^{N} P_{ik}Q_{kj} \geq 0\,, \tag{5.3}$$

since both P_{ik}, Q_{kj} are non-negative for all $i \in \{1, \ldots, N\}$ by the assumption that P and Q are stochastic matrices.

To show ii., we have to show that for all $j \in \{1, \ldots, N\}$

$$\sum_{i=1}^{N} (PQ)_{ij} = 1\,. \tag{5.4}$$

We verify this by expanding the matrix product: take $j \in \{1, \dots, N\}$ arbitrarily, then

$$\begin{aligned}\sum_{i=1}^{N}(PQ)_{ij} &= \sum_{i=1}^{N}\left(\sum_{k=1}^{N} P_{ik}Q_{kj}\right) = \sum_{i,k=1}^{N} P_{ik}Q_{kj} \\ &= \sum_{k=1}^{N}\left(Q_{kj}\sum_{i=1}^{N} P_{ik}\right) = 1\,,\end{aligned}$$

since for all $k \in \{1, \dots, N\}$, we have that $\sum_i P_{ik} = 1$ and for all $j \in \{1, \dots, N\}$, we have $\sum_k Q_{kj} = 1$ since P and Q are by assumption stochastic matrices. □

Proposition 5.3. *Define the norm $||w||_1 \equiv \sum_j |w_j|$. If P is a stochastic matrix, then for any vector y, it holds that*

$$||Py||_1 \leq ||y||_1\,. \tag{5.5}$$

Proof. We can verify this by a direct computation:

$$\begin{aligned}||Py||_1 = \sum_{j=1}^{N}|(Py)_j| &= \sum_{j=1}^{N}\left|\sum_{i=1}^{N} P_{ji}y_i\right| \leq \sum_{j,i=1}^{N} P_{ji}|y_i| \\ &= \sum_{i=1}^{N}\left(|y_i|\sum_{j=1}^{N} P_{ji}\right) = \sum_{i=1}^{N}|y_i| = ||y||_1\,,\end{aligned}$$

where we used both properties of the definition of a stochastic matrix. □

Corollary 5.3.1. *An eigenvalue λ of a stochastic matrix P satisfies $|\lambda| \leq 1$.*

Proof. Let P be a stochastic matrix, and λ an eigenvector of P with y a corresponding eigenvector. Then

$$||Py||_1 = ||\lambda y||_1 = |\lambda|\sum_{j=1}^{n}|y_j| \leq \sum_{j=1}^{n}|y_j|\,,$$

where we used Proposition 5.3 to obtain the last inequality. Hence $|\lambda| \leq 1$. □

Proposition 5.4. *Let P be a stochastic matrix. Then P has $\lambda = 1$ as an eigenvalue.*

Proof. Let P be an arbitrary stochastic matrix and suppose P is of dimensions $N \times N$. The $1 \times N$ row-vector $z = (1, 1, \dots, 1)$ is a left eigenvector of P with eigenvalue $\lambda = 1$. Indeed, note that for all $i \in \{1, \dots, N\}$, we have that

$$(zP)_i = \sum_{j=1}^{N} z_i P_{ij} = \sum_{j=1}^{N} P_{ij} = 1 = z_i\,,$$

such that $zP = z$. Hence z is an eigenvector of P with eigenvalue $\lambda = 1$. □

6 Detailed balance

Consider a system with three states with energies $E_1 < E_2 < E_3$. The dynamics are such that the system can make the transitions $1 \to 2$ with probability a, $2 \to 3$ with probability b, $3 \to 1$ with probability c or stay in the current state, where a, b, c are arbitrary rates for these transitions. Note that we require $0 < a, b, c < 1$. The stochastic matrix governing the evolution of this system is

$$P = \begin{pmatrix} 1-a & 0 & c \\ a & 1-b & 0 \\ 0 & b & 1-c \end{pmatrix} . \tag{6.1}$$

The dynamics of this system is ergodic: starting from 1, we have a non-zero probability to directly go to 2, and there exists a path from 1 to 3, which is $1 \to 2 \to 3$. From 2, there is a non-zero probabilty to go to 3, and there exists a path from 2 to 1, which is $2 \to 3 \to 1$. Starting from 3, there is a non-zero probability to go to 1, and there exists a path from 3 to 2, which is $3 \to 1 \to 2$. Hence any state in the system can be reached from any other state.

To verify if detailed balance is satisfied or not, we first compute the stationary state ω corresponding to this dynamics. This is the state that satisfied $P\omega = \omega$, and hence is an eigenvector of the matrix P with eigenvalue equal to one. In the previous section, we have proven that for any stochastic matrix P, such a vector always exists. Hence we can directly look for its components by solving the eigenvalue equation, which yields the system of equations

$$\begin{cases} (1-a)\omega_1 + c\omega_3 &= \omega_1 \\ a\omega_1 + (1-b)\omega_2 &= \omega_2 \\ b\omega_2 + (1-c)\omega_3 &= \omega_3 \end{cases} \iff \begin{cases} -a\omega_1 + c\omega_3 &= 0 \\ a\omega_1 - b\omega_2 &= 0 \\ b\omega_2 - c\omega_3 &= 0 \end{cases} . \tag{6.2}$$

It can be directly verified that $\omega = (c, \frac{ac}{b}, a)$ is a solution to this set of equations (remember that $b > 0$). The actual form of the components is irrelevant; the crucial point is that all components of the stationary state vector are different from zero, by the assumptions made on a, b, c. This information is sufficient to show that detailed balance is **not** satisfied for this dynamics. The problem here is that if we look again at the stochastic matrix responsible for the development of the dymanics, given in equation (6.1), we see that the zeroes in the matrix are not present in a symmetric manner. Indeed, take $\mu = 2$ and $\nu = 1$; then $P(2 \to 1) = 0$, while $P(1 \to 2) \neq 0$, and hence the condition for detailed balance

$$\omega_\nu P(\nu \to \mu) = \omega_\mu P(\mu \to \nu) , \tag{6.3}$$

is not satisfied for this choice of μ and ν, since the left hand side is different from zero, while the right hand side is equal to zero.

In the lecture notes, we saw that detailed balance is a sufficient condition to get a Boltzmann distribution for the equilibrium of the dynamics. Here we show that even though detailed balance is not satisfied, we can still choose the rates a, b, c in such a manner that the equilibrium distribution is a Boltzmann distribution, proving that detailed

balance is in fact not a necessary condition. In order to have a Boltzmann distribution, we require that the probability of finding the system in state i is given by

$$p_i = \frac{e^{-\beta E_i}}{Z}\,, \tag{6.4}$$

where $Z = \sum_{i=1}^{3} e^{-\beta E_i}$ is the partition function. Looking back at our result for the stationary state ω, we find that we can achieve this by setting

$$a = \frac{e^{-\beta E_3}}{Z}, \quad b = \frac{e^{-\beta(E_1+E_3-E_2)}}{Z}, \quad c = \frac{e^{-\beta E_1}}{Z}\,. \tag{6.5}$$

Since then we indeed have

$$\begin{aligned}
\omega_1 &= c = \frac{e^{-\beta E_1}}{Z} \\
\omega_2 &= \frac{ac}{b} = \frac{e^{-\beta E_2}}{Z} \\
\omega_3 &= a = \frac{e^{-\beta E_3}}{Z}\,,
\end{aligned}$$

such that the equilibrium distribution of the dynamics is a Boltzmann distribution.

7 Ising Model: Uniform sampling

In the 2D Ising model, spins $s_i = \pm 1$ are located at the vertices of a $N \times N$ square lattice. We will often refer to a spin value $+1$ as spin up, while spin value -1 is called spin down. The energy of such a configuration of spins is given by

$$E\left(\{s_k\}\right) = -J \sum_{\langle ij \rangle} s_i s_j\,, \tag{7.1}$$

where the summation is over nearest neighboring spins on the lattice (for a 2D lattice, there are 4 such nearest neighboring spins).The coupling constant J is set to 1 throughout this and the next section.

In this section, we approach the Ising model with a Monte Carlo algorithm that uses uniform sampling as a demonstration that this is an inefficient method. We generate a lattice using the helical boundary conditions, as described in full detail in the lecture notes. The method of uniform sampling to generate configurations of spins is also described in the lecture notes. Each run of the algorithm yields a vector ω, which is of length N^2, representing the 2D lattice of spins. Next, we discuss the method of Boltzmann sampling with a hit-and-miss method, which will also prove to be inefficient for most ranges of the temperature. Finally, we investigate the technique of reweighting, which will then be compared with the output of the Metropolis algorithm in the next section.

7.1 A few remarks on the Ising model

Before delving into the algorithm and its outcomes, we make some comments about the Ising model which will be used later on. First of all, to compute the energy of a configuration of spins, we can iterate over all spins s_i in the lattice, and compute $s_i s_j$, for nearest neighbours s_j in equation (7.1) for each spin. However, it is clear that this method will count all pairs of spins that contribute to the energy exactly twice, and hence we have to divide the end result by two to obtain the correct energy. This way of computing the energy is very simple to implement in an algorithm, but it is immediately clear that this can be done more efficiently: we now waste a lot of computer resources to compute terms which are redundant for the calculation. We will, however, not try to optimise

It is also useful to note that the ground state of the Ising model is $E_0 := -2N^2$. Indeed, it is clear from equation (7.1) that the ground state corresponds to a lattice where all the spins are aligned with each other ($s_i = +1$ or $s_i = -1$ for all i). The energy of this configuration can be computed by the method described above, and since all terms equal $s_i s_j = 1$, this yields $E_0 = -4N^2/2 = -2N^2$.

Check that the following is true

Finally, we remark that while the ground state obtains an energy $E_0 = -2N^2$, there does *not* exist a state of spins for which the energy is $+2N^2$. To show this, note that $2N^2$ is the maximum energy that a lattice could in principle obtain, and this is achieved when for each spin we pick in the lattice, all the neighbouring spins are unaligned with respect to that spin such that all terms in equation (7.1) are positive. Hence the only possible configurations[1] are drawn in Figure 7.1 using the following reasoning. We draw the first row by alternating between plus and minus, and repeat for the next row, but flipping all the spins of the first row. This is repeated until the lattice is filled. Of course, the direction of the final spin on each row depends on whether N is even or odd, as shown in Figure 7.1.

Figure 7.1: Configuration of spins in a $N \times N$ lattice in which neighbouring spins are maximally unaligned inside the lattice. *Left*: N is even, *Right:* N is odd.

This is the only configuration for which all the spins inside the lattice are unaligned with their neighbouring spins, as is easily checked. However, the problem arises at the

[1] Flipping all the spins in the lattices drawn in Figure 7.1 would in principle give another diagram, but the energy of that diagram is the same. For the energy, the labels + and − do not matter, only the alignment between neighbouring spins is important.

boundaries. Suppose we take periodic boundary conditions. Then in fact the lattice drawn on the left, for N even, is a configuration of spins for which all the spins are unaligned with their neighbours. This is not true in case N is odd: spins at the boundaries of each row are aligned with one of their neighbours at their left or right hand side. Hence the energy cannot reach the maximum of $2N^2$, since there are contributions which are negative.

If we would rather take helical boundary conditions, there is again a problem at the boundaries. The spins at the lower left corner and upper right corner are neighbours. Indeed, the spin at the upper right corner is 'to the left' of the spin in the lower left corner if helical boundary conditions apply. For both even and odd N, we see that these two spins are such that again the maximum $2N^2$ cannot be reached.

7.2 Uniform sampling

The method of uniform sampling is easy to implement: for each lattice site, we 'flip a coin', determining whether the spin at that lattice site is up or down. The result, as mentioned before, is stored in a vector of length N^2.

For a fixed value of N, e.g. $N = 16$, we generate the histogram of energy values that we obtain after running the uniform sampling algorithm 10^5 times. Instead of looking at the energy E, we look at the intensive variable $e = E/2N^2$, called the energies per bond. The histogram is shown in Figure 7.2.

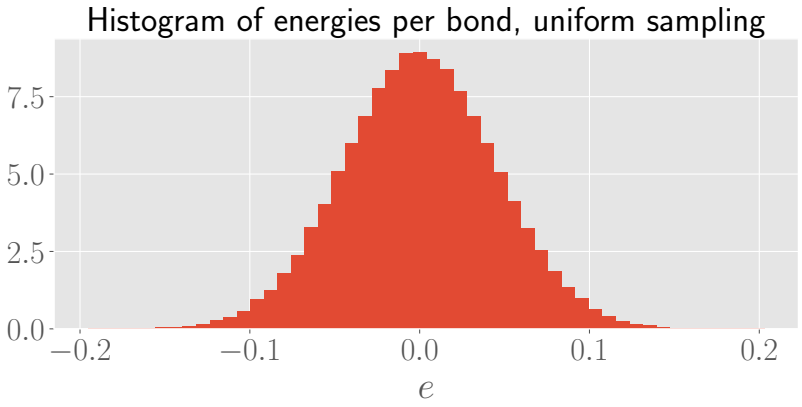

Figure 7.2: Histogram of $e = E/2N^2$, obtained via uniform sampling and 10^5 runs of the algorithm.

7.3 Hit-and-miss method

The hit-and-miss method depends on the temperature T: as before, a random configuration of spins is generated, and the energy of the system E is computed. Next, we generate a uniform random number r in the interval $[0, 1]$. If we have $r \leq \exp(-\beta(E - E_0))$, where $\beta = 1/T$ (we set $k_b = 1$), then this is a 'hit' and the configuration is accepted.

We are interested in the efficiency of the hit-and-miss algorithm in function of the temperature T. For this, we calculate for various temperatures $T_1 = 10^5$, $T_2 = 10^3$, $T_3 = 10^2$ and $T_4 = 10$ the percentage of accepted lattice configurations over the total generated configurations. We also look at the dependence of this percentage on the size of the lattice.

	T_1	T_2	T_3	T_4
$N = 4$	100.0	97.6	73.9	5.1
$N = 8$	99.7	89.2	28.5	0.0
$N = 12$	99.6	73.9	6.4	0.0
$N = 16$	99.4	61.3	0.9	0.0

Table 7.1: Percentages of hits for various temperatures T (defined in the text) and various sizes of the lattice N.

As expected, at high temperatures (low β), the probability to accept a configuration of spins is relatively high. For $T_1 = 10^5$, we have $\beta = 10^-5$ and a high probability to accept the configuration, while for larger β, this becomes smaller. Recalling that the critical temperature for the Ising model is $T_C \approx 2.27$, and hence the interesting physics happens around low temperatures, we can conclude that he hit-and-miss algorithm performs very poorly.

what about the dependence on N?

7.4 Reweighting technique

Since the hit-and-miss method performs poorly over a large range of temperatures, we discuss the reweighting technique. From the histogram we obtained by the uniform sampling method, $P(E)$, we generate for a given temperature T another histogram $P_T(E) = \exp(-E/T)P(E)$. In Figure 7.3, we plot the histogram of energies of 10^5 lattices with $N = 16$ using the uniform sampling.

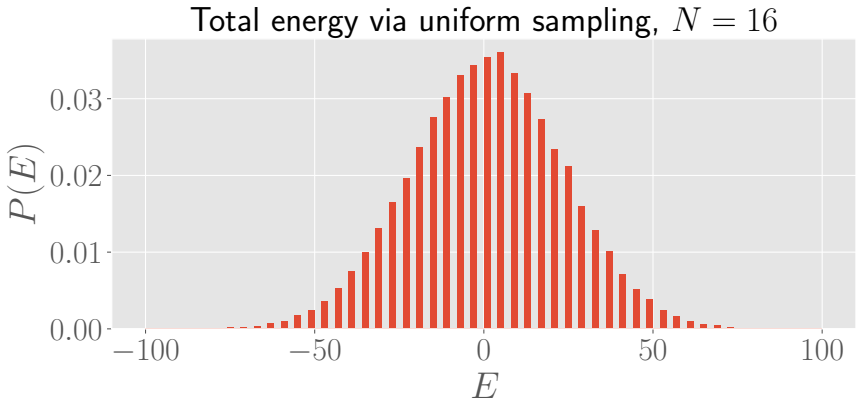

Figure 7.3: Histogram of the total energy of 10^5 lattices using uniform sampling.

This histogram is used to obtain the reweighted histograms $P_T(E)$ for temperatures $T = 10^5$, $T = 10^3$, $T = 10$ and $T = 2$, shown in Figure 7.4.

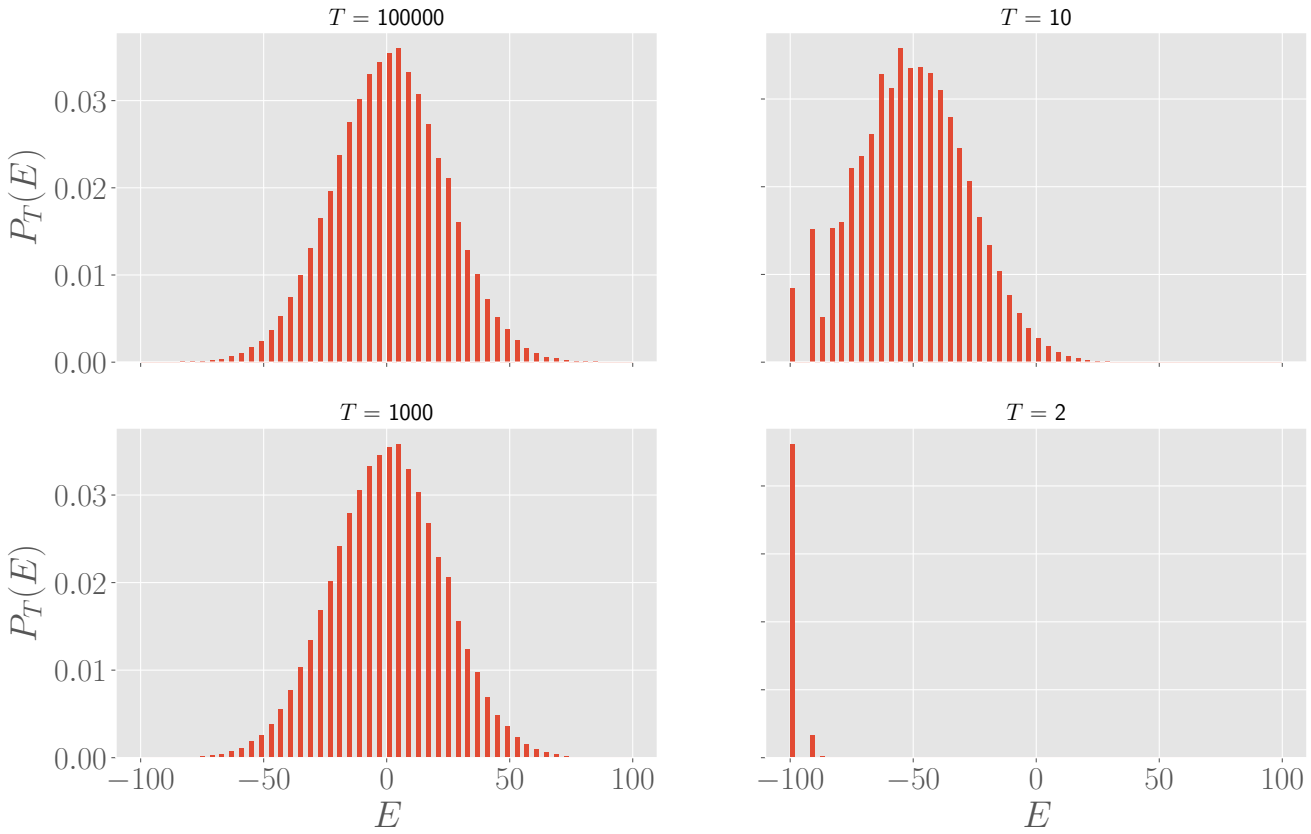

Figure 7.4: Histograms of the total energy at four temperatures, obtained via the reweighting technique.

It is clear that this method does not seem very adequate either at generating histograms of the total energy at different temperatures. There is almost no effect for the two highest temperatures, since the energies of the uniform histogram bounded in absolute value by 100, such that $P(-E/T)$ ranges from around 0.9 to 1.1 for $T = 10^3$, which causes no dramatic changes to the histogram. This is different for lower temperatures: the weights of the lowest values of the histogram are large, while energies around zero become negligible.

8 Metropolis algorithm for the Ising model

We continue the analysis of the Ising model from the previous section and now implement the Metropolis algorithm, where we sample spin configurations through a Markov chain. Let μ and ν represent two configurations of spins. The transition probability $P(\mu \to \nu)$ is non-zero if and only if the two configurations differ only by a single spin.

Given an initial configuration of the spins, randomly generated, we select one of the spins at random, and then suggest to flip it. We compute the change in energy

$\Delta E = E_f - E_i$ that would arise to flipping that spin. If $\Delta E \leq 0$, i.e. if the system would evolve towards a lower energy configuration, we accept the flip. If $\Delta E > 0$, we still accept the suggested flip if $r \leq \exp(-\Delta E/T)$, where r is a uniform random number generated in the interval $[0, 1]$. This is repeated a large number of times. We use the Metropolis algorithm to investigate

i. magnetisation per spin of a spin $m = \frac{1}{N^2}\sum_i s_i$,

ii. the equilibration time,

iii. the histogram of total energy at low temperatures,

iv. the auto-correlation function $\chi(t) = \frac{1}{t_f}\sum_{s=\tau_{eq}}^{\tau_{eq}+t_f}(m(s) - \langle m\rangle)(m(s+t) - \langle m\rangle)$.

Recall that the critical temperature of the ising model is $T_C \approx 2.269$, and that the exact value of the magnetisation is given by

$$m(T) = [1 - \sinh^{-4}(2J/T)]^{1/8}\,, \tag{8.1}$$

and that throughout this report, we take $J = 1$.

8.1 Efficiency of the Metropolis algorithm

We would like to make our code for the Monte Carlo algorithm as efficient as possible, since this code will be repeated a large number of times. In the Metropolis algorithm described above, we need the difference in energy ΔE between two different configurations. However, since only one spin differs between these configurations, it is sufficient to compute the change in energy that arises because of this flipped spin. Indeed, all other spins remain unchanged, and will give no difference in energy. Therefore, we only need to compute the contributions toward the energy due to this spin and its four nearest neighbours, instead of computing the total energy in the lattice.

If $\Delta E > 0$, we need to compute an exponential to determine whether or not we accept the changed spin. This requires a certain amount of polynomial time, but this can be sped up if we compute the possible outcomes of this exponential function beforehand, which can be done since there are only finitely many $\Delta E > 0$. Indeed, we claim that the only possible values of $\Delta E > 0$ are $\Delta E = 4$ and $\Delta E = 8$.

To prove this claim, recall that ΔE is only determined by the selected spin and its neighbours. Before the spin is flipped, the energy configuration is

$$E_i = (-1)\cdot\chi_a + (+1)\cdot\chi_u\,, \tag{8.2}$$

where χ_a, respectively χ_u, is the number of neighbouring spins that are aligned, respectively unaligned with the selected spin. By flipping the spin, the energy of the final configuration is

$$E_f = (-1)\cdot\chi_u + (+1)\cdot\chi_a = -E_i\,, \tag{8.3}$$

and so we have $\Delta E = E_f - E_i = -2E_i > 0$. Hence we have $E_i < 0$, such that $\chi_a > \chi_u$. Since $\chi_a + \chi_u = 4$, the only possibilities for the initial configuration are $\chi_a = 3, \chi_u = 1$

or $\chi_a = 4$, $\chi_u = 0$. The first case has an energy $E_i = -2$, the second one an energy $E_i = -4$, and since $\Delta E = -2E_i$, this proves that if $\Delta E > 0$, then we have $\Delta E = 4$ or $\Delta E = 8$. This proves our claim.

Hence given a temperature for the system T, we compute and store the values of $\exp(-4/T)$ and $\exp(-8/T)$ beforehand to make the algorithm more efficient. We know from the lectures that the Ising model has critical slowing down. Since we will simulate at $T = 2$ and $T = 2.2$, which is close to the critical temperature, we will start from a lower temperature and let m converge close to the exact value $m(T)$ for a temperature T of interest. We can then start the algorithm again with the correct temperature, and start from the last spin configuration of the previous run.

8.2 Magnetisation per spin

For $T = 2$ and $N = 50$, we compute the magnetisation per spin, defined above, at each Monte Carlo time step. The step size is one sweep, which is equal to N^2 iterations of the Metropolis algorithm. The first 200 sweeps are done with a temperature $T = 0.2$, after which the Metropolis algorithm is applied for $T = 2$ and 500 iterations. The result is shown in Figure 8.1. From the close-up on the right hand side in this figure, we can deduce that $\tau_{eq} \approx 100$ sweeps, starting from the final configuration at $T = 0.2$.

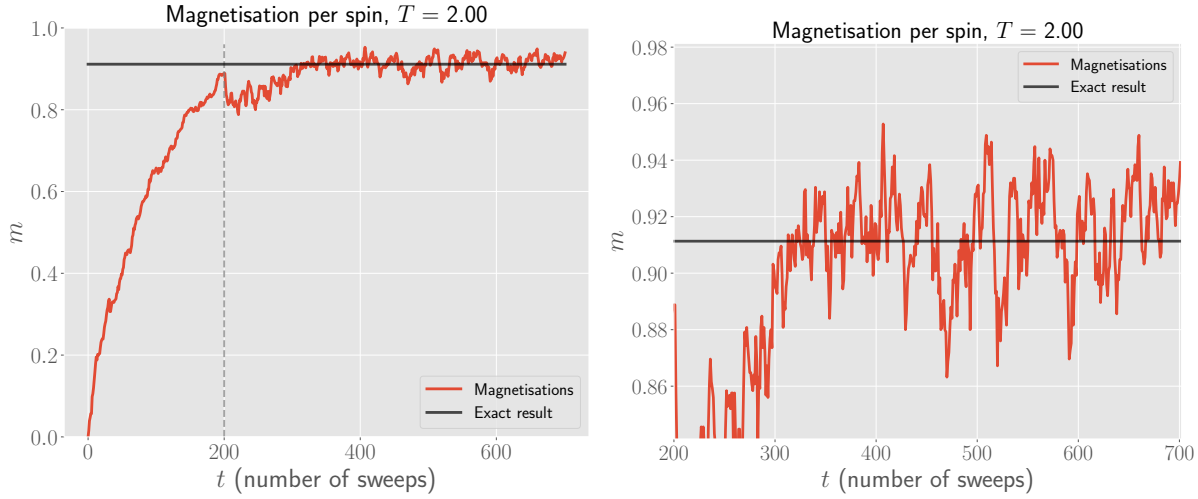

Figure 8.1: Magnetisation per spin in function of number of sweeps. *Right:* Close-up of the plot on the left hand side.

8.3 Histogram of total energy

In the previous section, we saw that both the hit-and-miss method and the reweighting technique were not suited to generate histograms of the total energy of the Ising model at low temperatures. Indeed, the histogram for $T = 2$ in Figure 7.4 did not provide us with much insights. We now apply the Metropolis algorithm to generate such histograms.

We start from the same final configuration as in the previous subsection. We know that it takes around 100 sweeps to let the system equilibrate if we start from this configuration. After the equilibrium is settled, we compute the energy per bond e of the lattice at each sweep. The resulting histogram is shown in Figure 8.2.

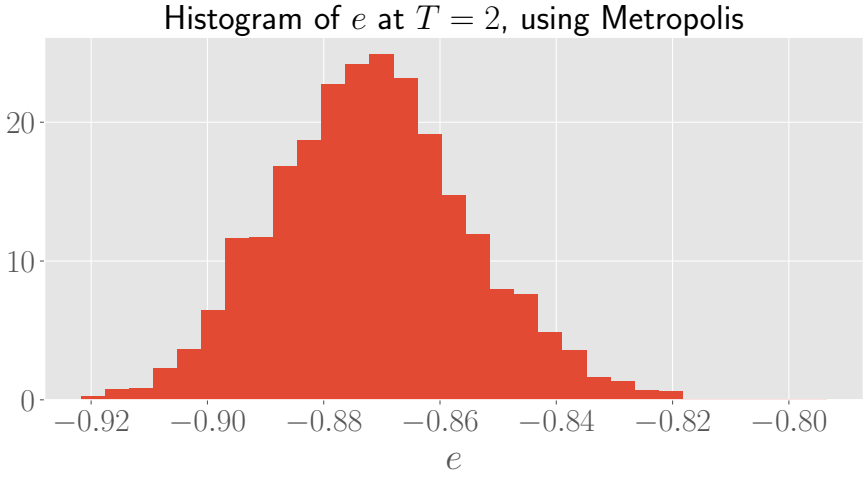

Figure 8.2: Histogram of the energy per bond e of the lattice

The Metropolis algorithm seems to yield a correct histogram of the energies at low temperatures. The interpretation of the histogram above is that the equilibrium of the system is at some configuration with an energy close to the ground state (for which $e = -1$), and the Metropolis algorithm causes oscillations around this equilibrium. This can be explained from the physics behind the Ising model. Configurations close to the ground state have most of their spins aligned, and hence a magnetisation close to 1. For $T = 2$, equation (8.1) yields an exact magnetisation of $m(2) \approx 0.911$. Hence we indeed predict an equilibrium configuration with an energy around $0.9E_0$.

8.4 Autocorrelation function

We now look at the autocorrelation function for $T = 2.2$ and $N = 50$. For this, we take the final configuration of a run which has reached equilibrium for $T = 2$ in the previous exercises, and converge towards the equilibrium distribution for $T = 2.2$. Next, we sample the magnetisations after equilibrium is reached, and compute $\chi(t)$, as defined above, for a large value of t_f (for details: see the notebook). The obtained values of $\chi(t)$ are normalised by dividing by $\chi(0)$. The results for one such run are shown in Figure 8.3.

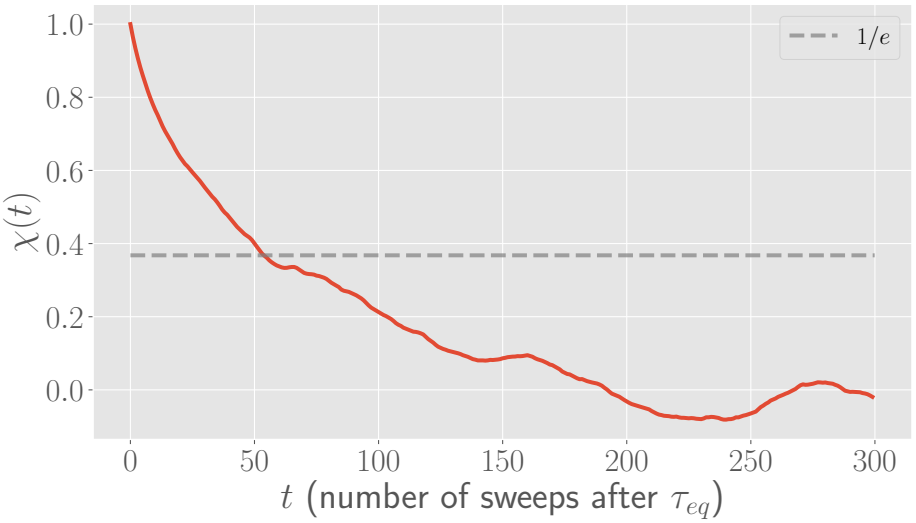

Figure 8.3: Plot of the autocorrelation function $\chi(t)$ for $T = 2.2$ and $N = 50$ after equilibrium is reached.

From the lecture notes, we know that $\chi(t) \sim e^{-t/\tau}$, where τ is called the autocorrelation time. We can estimate this from the above plot by searching for the time t at which $\chi(t) = 1/e$: we estimate this to be around 50 sweeps. We can fit an exponential curve using Scipy through our $\chi(t)$ data, ignoring the tail of the above plot, since these fluctuations are likely due to noise in running our algorithm. This fit estimates $\tau \approx 66$ sweeps as optimal parameter (more details in the notebook). We can check if this is a reasonable estimate by looking at the magnetisations on which our $\chi(t)$ calculations are based. A particular close-up of these magnetisations is shown in Figure 8.4, and seems in accordance with an autocorrelation time of approximately 60 sweeps.

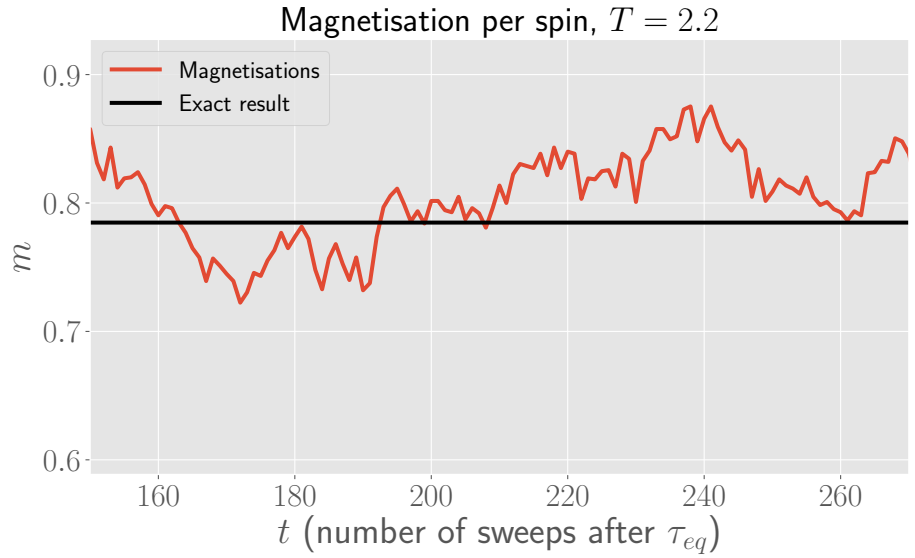

Figure 8.4: Close-up of magnetisations after equilibration for $T = 2.2$.

9 Lotka-Volterra Model

The Lotka-Volterra model describes the evolution of two populations X_1, X_2, which are called the *prey* and the *predator*. The relevant reactions are

$$X_1 \xrightarrow{k_1} 2X_1$$
$$X_1 + X_2 \xrightarrow{k_2} 2X_2$$
$$X_2 \xrightarrow{k_3} \emptyset\,,$$

where $k_i, i = 1, 2, 3$ are the rates. We set $k_1 = 1$, $k_2 = 0.005$ and $k_3 = 0.6$ and start from the intial state with 100 preys and 20 predators. As mentioned in the lecture notes, it is shown that the rate equations for the Lotka-Volterra system are

$$\frac{\mathrm{d}}{\mathrm{d}t}X_1 = k_1X_1 - k_2X_1X_2 \tag{9.1}$$
$$\frac{\mathrm{d}}{\mathrm{d}t}X_2 = k_2X_1X_2 - k_3X_2\,. \tag{9.2}$$

Hence the reaction rates are

$$a_1 = k_1X_1, \quad a_2 = k_2X_1X_2, \quad a_3 = k_3X_2\,. \tag{9.3}$$

As a last remark, we recall that in the lecture notes, it is shown that the non-zero stationary solution is given by

$$X_1^{(s)} = \frac{k_3}{k_2}, \quad X_2^{(s)} = \frac{k_1}{k_2}\,. \tag{9.4}$$

For our set-up, we have $X_1^{(s)} = 120$ and $X_2^{(s)} = 200$. In the lecture notes, it is also shown that the system has oscillatoric solutions around the non-zero stationary solution.

Looking back at equations (9.1) and (9.2), we see that there are two so-called *absorbing states* for the Lotka-Volterra system. The first one occurs when $X_1 = 0$, since then $a_1 = 0$ and $a_2 = 0$, and only the third reaction can take place. Since this reaction causes the population X_2 to decrease, this will eventually lead to $X_2 = 0$ after a finite time. Our predator-prey analogy is then that when all the prey is extinct, the predators will eventually also end up being extinct, since they have no food. This scenario can therefore be called the 'extinction scenario', since after both populations will end up going extinct.

The second absorbing state is when $X_2 = 0$. In that case, $a_2 = 0$ and $a_3 = 0$, such that only the first reaction can take place. This will cause the population X_1 to increase indefinitely from that moment onwards. This scenario can be dubbed the 'victory of the prey'. In short, the absorbing state $X_1 = 0$, $X_2 = 0$ is reached if for some $t_f > 0$, we have $X_1(t_f) = 0$, while the absorbing state $X_1 \to +\infty$, $X_2 = 0$ is reached if for some $t_f > 0$, we have $X_2(t_f) = 0$.

9.1 The Gillespie algorithm for Lotka-Volterra

We implement the Gillespie algorithm, discussed in the lecture notes, for the Lotka-Volterra system, and illustrate the algorithm by plotting a trajectory in the (X_1, X_2)

plane, shown in Figure 9.1. Note that both plots have the same initial condition and are obtained with the same algorithm, but clearly have a different behaviour. The system tends to oscillate around the non-zero stationary solution, as argued above, change and relocate this but in most of the cases it ends in the "extinction" of one of the two species, i.e., for some $t_f > 0$, either $X_1(t_f) = 0$ or $X_2(t_f)$. This point is denoted as the 'extinction point' in the plots below. The term 'extinct' is explained in section 9.3. Moreover, it seems that the population of the prey goes extinct more often than the population of the predators. We will return to this feature in more detail in section 9.3.

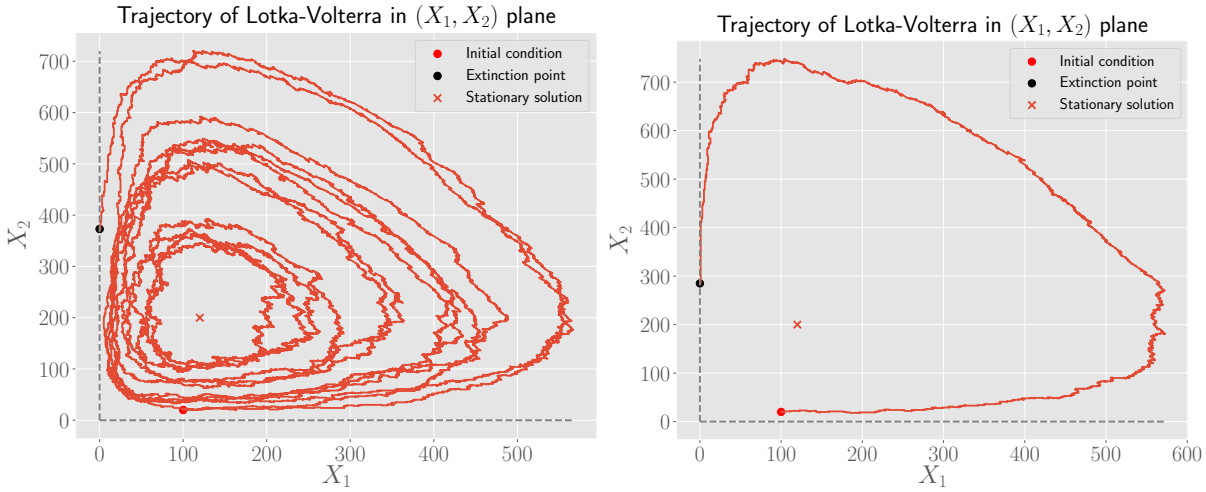

Figure 9.1: Two different trajectories of the Lotka-Volterra system in the (X_1, X_2) plane obtained via the Gillespie algorithm.

In Figure 9.2 below, we also plot X_1 and X_2 as a function of time for two different trajectories of the system. In both cases, the system ends in $X_1(t_f) = 0$. Again we note that the behaviour of the trajectory can vary between different runs: the plot on the left represents a lot of oscillations around the stationary solution. This behaviour is similar as the one in the plot on the left in Figure 9.1. Likewise, the plot on the right has not a single oscillation, and its behaviour is similar to the trajectory of the plot on the right in Figure 9.1. The oscillations of the Lotka-Volterra system are discussed in the next section.

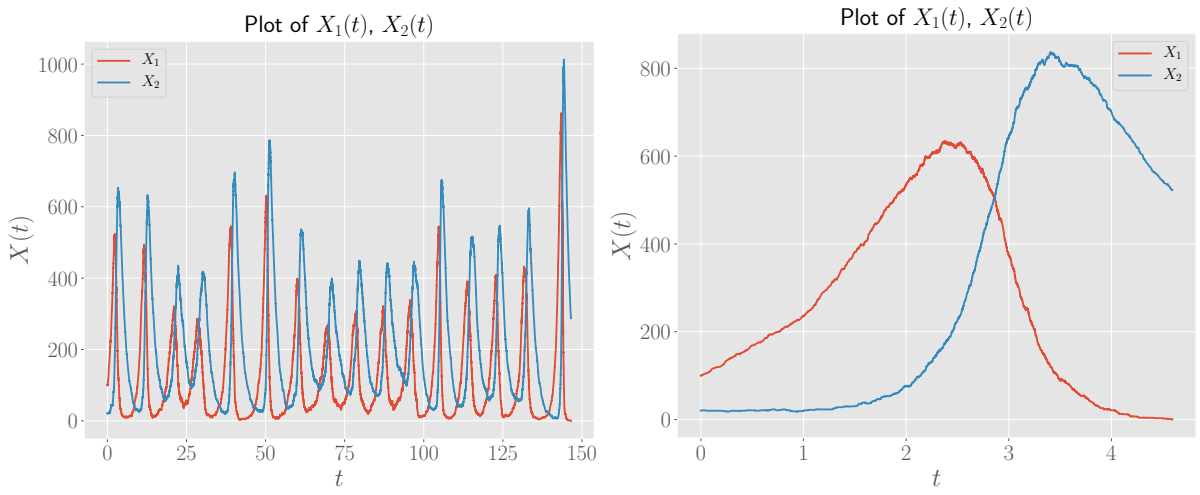

Figure 9.2: Two different plots of $X_1(t)$ and $X_1(t)$. In both cases, the system ends in $X_1(t_f) = 0$.

9.2 Oscillations of the Lotka-Volterra system

As already indicated in the figures above, the system performs oscillations around the stationary solution, but the exact number of such oscillations can vary wildly. In some cases, only one oscillation is present, while in some rare occurrences, as much as 31 oscillations could be seen. We write a function in Python which estimates the number of oscillations in the system. This allows us to efficiently determine the number of oscillations in a large number of runs to estimate its average. For this function, we say that X_1 is 'large' if it is larger than 100, and we say it is 'small' if it is smaller than 70. These values are arbitrarily chosen, but they seem to indicate whenever the graph is tending towards a peak, or whenever it is going through a valley quite well. We also remark that the values in those valleys are most often low, while the peaks vary wildly in height from around 100-120 to more than 1000. In some runs with a large number of oscillations ($\gtrsim 30$), some of the oscillations are so short with such small peaks that the function is not able to notice them. We believe this effect is negligible when averaging over a large number of runs.
The function essentially takes the data of $X_1(t)$, and searches when the data goes from 'large' to 'small' or vice versa. Whenever subsequently a transition from large to small, and back to large again is noticed, the count of number of oscillations increases with one. After testing the function a few times, it seems to guess the number of oscillations correctly in almost all of the runs.

If we let the Gillespie algorithm run 1000 times and determine the number of oscillations for each run, it seems that the average lies between 3 and 4. When rounded to the nearest integer, the average number of oscillations seems to yield 3 in most cases.

It is important to note that this function only considers oscillations in the $X_1(t)$ data. However, the trajectories, as shown on the left in Figure 9.2, indicate that $X_2(t)$ follows a similar pattern in time, with a short delay with respect to $X_1(t)$. Hence we limit ourselves to determining the number of oscillations in the $X_1(t)$ data, in order to limit the amount of computation time. Moreover, as is clear from the discussion above, the labels 'large'

and 'small' were chosen rather empirically: one could propose a more precise condition to determine when an oscillation occurred, if one would wish a more reliable estimation.

9.3 Absorbing states of the Lotka-Volterra system

We now investigate the absorbing states of the Lotka-Volterra system. A short script in Python is written which counts how often the system ends in one of the two absorbing states before a certain time limit is reached. The time limit is chosen to be very large. discuss why.

In one of those runs, where we let the algorithm run a 1000 thousand times, 97.90% of the runs ended in the extinction of X_1, and 2.10% ended in the extinction of X_2. Running the code block several times, we see that at least $\approx 95\%$ of the runs end with the extinction of X_1. Hence we conclude that only very rarely, the predators go extinct. It seems that the system always ends in one of the two absorbing states.

So we can conclude that our Lotka-Volterra system (i.e., the Lotka-Volterra dynamics with our chosen initial condition) oscillates around the non-zero stationary solution on average three to four times, before ending in one of the absorbing states, which is typically the state in which the preys go extinct.

Lastly, we consider the distribution of t_f, the final time or 'extinction time', as defined above. We focus on times $t_f \leq 200$ to determine the distribution. Plotting a histogram, it seems the times decrease exponentially. A general exponential distribution is given by

$$F(x|\lambda) = 1 - \exp\left(-\lambda x\right) , \tag{9.5}$$

where λ is the rate parameter. Recall that the mean of the exponential distribution is equal to the reciprocal of the rate parameter. Therefore we plot an exponential cumulative density function on top of our histogram, estimating the rate parameter by taking the sample mean. This is justified, since the sample mean is a maximum likelihood estimator, and hence can be considered a good estimator. The histogram after 2000 iterations is shown in Figure 9.3. Overall, the data indeed seems to agree quite well with an exponential distribution.

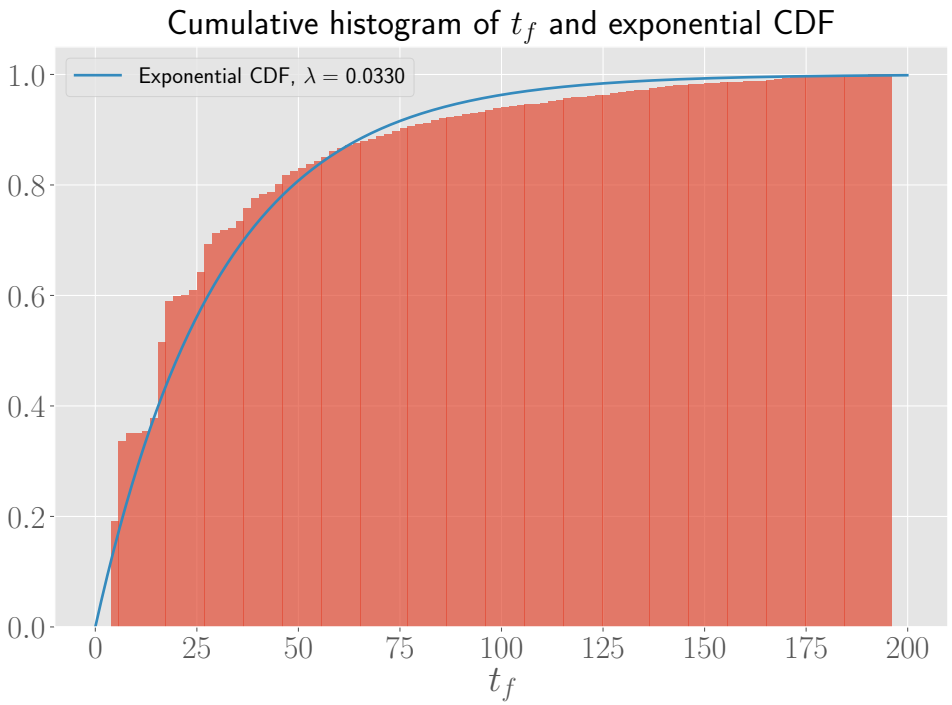

Figure 9.3: Cumulative histogram of 2000 extinction times. The exponential CDF with sample mean as estimator for the rate parameter shown in blue.