MP468 Computational Physics II Notes

John Brennan[†]

[†]Department of Theoretical Physics, Maynooth University

Contents

1 Part A: Random numbers and Monte Carlo methods

1.1 Introduction

The first topic we cover in this course is how to produce random real numbers with different distributions. To motivate the topic, we begin by briefly outlining how random numbers can be used to solve integrals. Let's suppose we want to integrate the function f over the interval $[a, b]$.

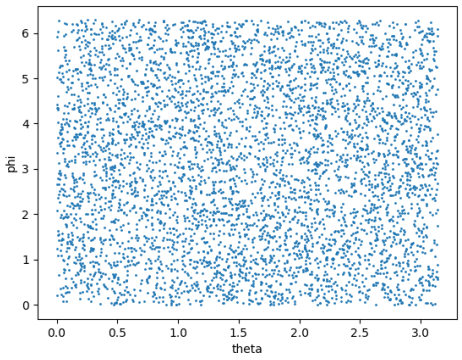
$$I = \int_a^b f(x)dx. \tag{1}$$

In MP368, we would compute this integral by first dividing the interval $[a, b]$ into N subintervals of length $dx = \frac{b-a}{N}$, for some large number N, and then approximate the area under the curve of f and over each subinterval as the area of a rectangle or trapezium. The integral was then calculated by adding up all of these approximations.

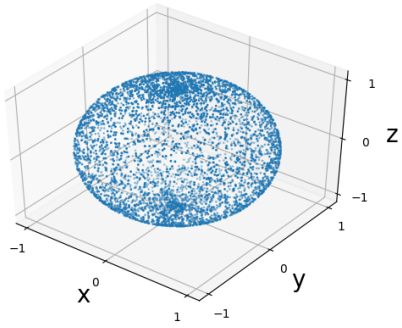
Another way to calculate the integral is to pick a large number N of random points x_i uniformly distributed in the interval $[a, b]$ and evaluate the function f at each of them. Then we can estimate the average value of f over the interval $[a, b]$ and the integral can be calculated using the fact that the area under the curve is equal to the length $b - a$ times the average value of f over $[a, b]$.

$$I \approx (b - a) \sum_{i=1}^{N} \frac{f(x_i)}{N} \tag{2}$$

An essential part of this method (known as a Monte Carlo method) is generating random numbers in the interval $[a, b]$ with the right distribution. For example, if the random numbers we generate, in $[a, b]$, happen to be distributed in such a way that they cluster about a particular point c in $[a, b]$, then our estimate for the average of f will be skewed and our evaluation of I won't be accurate. For the Monte Carlo method just described, we want our randomly chosen points, in the domain we're integrating over, to be uniformly distributed. For us, generating numbers uniformly distributed over a finite interval is easy (the numpy function numpy.random.rand generates a random number x in the interval $[0, 1)$ and so we can get a uniformly random number y in $[a, b)$ by taking $y = (b - a)x + a)$. However, in general, we might want a more complicated distribution of numbers. For example, let's suppose we want to integrate a function over the surface of a sphere using the Monte Carlo method described above. To do this, we'll need to generate a large collection of random points on the sphere that are uniformly distributed. To achieve this one might parametrise the sphere using spherical coordinates $(\theta, \phi) \in [0, \pi) \times [0, 2\pi)$ and then generate N uniform random numbers in $[0, \pi)$ and N uniform random numbers in $[0, 2\pi)$ resulting in N random points (θ_i, ϕ_i) on the sphere. However, the resultant collection of random points on the sphere will not be uniformly distributed over the sphere. The following python code demonstrates this by producing such a collection



(a)

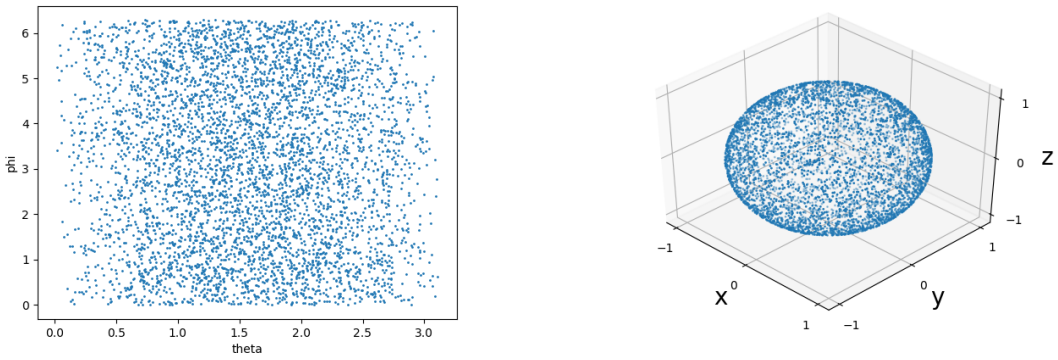


(b)

Figure 1

of points on the sphere and plotting them as shown in Fig. 1a and Fig. 1b. In Fig. 1a we see the generated collection of points are uniformly distributed in the coordinate patch $[0, \pi) \times [0, 2\pi)$ while in Fig. 1b we see the points are not uniformly distributed over the sphere with points clustering around the north and south poles of the sphere.

```
import numpy as np
import numpy.random as ran
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

N = 5000                        # Number of points

theta = np.pi*ran.rand(N)       # uniform points between 0 and pi

phi   = 2*np.pi*ran.rand(N)     # uniform points between 0 and 2pi

# 3 dimensional coordinates
x = np.sin(theta)*np.cos(phi)
y = np.sin(theta)*np.sin(phi)
z = np.cos(theta)

# plot points on the sphere
fig = plt.figure(1)
ax  = fig.gca(projection='3d')
ax.scatter(x,y,z,s=1)

# plot points on the coordinate patch
plt.figure(2)
plt.plot(theta,phi,'o',markersize=1)
plt.xlabel('theta')
plt.ylabel('phi')
```



(a) (b)

Figure 2

The clustering around the north and south poles of the sphere is an affect of the coordinate system used. Recall the area element of the sphere in these coordinates is $d\Omega = \sin(\theta)d\phi d\theta$ which has its largest values at $\theta \approx \pi$ (the equator) and has its lowest values at $\theta \approx 0, 2\pi$ (the north and south poles). This means a small square of the coordinate patch, with area A, is mapped to a small patch of the sphere whose area is $\sin(\theta)A$. Hence, if the average density of random points within a small square of the coordinate patch is ρ, the average density in a patch of the sphere becomes $\rho \div \sin(\theta)$, i.e. the density near the poles is larger than the density of points near the equator. To rectify this, we need to generate random points (θ_i, ϕ_i) in the coordinate patch whose probability of appearing in a small square decreases as the square moves away from $\theta \approx \pi$ (the equator) the same way the area of the square decreases when it's mapped to the sphere. This can be achieved by using the following line of code to generate numbers θ_i instead of the one used previously on line 8 (This line of code may seem arbitrary but we will show why this works in the next section).

```
1 # non-uniform points between 0 and pi
2 theta = np.arccos( 1-2*(ran.rand(N)) )
```

Replacing the line of code used before which generated random numbers θ_i (line 8) with the above line of code and running it will produce the figures shown in Fig. 2a and Fig. 2b. We now see the density of points in the coordinate patch drops significantly near the edges where $\theta \approx 0, 2\pi$ and the clustering of points near the north and south poles has disappeared.

1.2 Random numbers

The first thing to note about random numbers generated on a computer is that they're not actually random. The only way to generate truly random numbers is by measuring physical processes like flipping a coin, throwing a dice, thermal fluctuations in the environment, quantum fluctuations of particles or chaotic systems such as the weather. In a typical computer you can't do any of these, all you have is a processor

that can execute a list of predefined instructions. For that reason, most programming languages use arithmetic to produce sequences of numbers which *look* random and are uncorrelated (meaning there's no obvious pattern to successive numbers). Such numbers are called pseudo-random numbers. Since pseudo-random numbers are generated by a deterministic algorithm, a sequence of pseudo-random numbers is completely determined by the initial number of the sequence (usually called the seed) and will eventually start repeating itself after a finite number terms. For the rest of these notes we'll refer to pseudo-random numbers as random numbers.

A probability density on $\mathbb{R}$ is a real function P which is non-negative everywhere such that

$$\int_{-\infty}^{\infty} P(x)dx = 1. \tag{3}$$

If X is a random variable with distribution P, then the probability that X takes a value in the subset $A \subset \mathbb{R}$ is given by

$$\text{Prob}_X(A) = \int_A P(x)dx. \tag{4}$$

In *numpy.random* we have access to a number of different numerical routines for generating random numbers under a variety of different distributions. Two notable functions are the *rand* and *randn* functions. The function *rand* generates uniformly distributed random numbers between 0 and 1, i.e. it generates a real number under the distribution

$$P(x) = \begin{cases} 1 \text{ if } 0 \leq x < 1 \\ 0 \text{ otherwise} \end{cases}, \tag{5}$$

The function *randn* generates a real number under the normal (or Gaussian) distribution:

$$N(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right). \tag{6}$$

It often happens in practise that we need to generate numbers under a distribution that isn't offered by any of the functions in numpy (or which ever numerical package we're using). We'll now cover two methods, known as the transformation method and the rejection method, which will give us the ability to generate random numbers under any distribution.

1.2.1 The Transformation Method

The transformation method is a way of generating random numbers whose premise we introduce by answering following question: If X is a random variable with distribution P and $f : \mathbb{R} \to \mathbb{R}$ is a function, how is the random variable $Y = f(X)$ distributed over $\mathbb{R}$? To answer this, we first note that the probability of X being in the interval (a, b) must be equal to the probability of Y being in the interval $(f(a), f(b))$. Hence, we have

$$\text{Probability } X \in [a,b] = \text{Probability } Y \in [f(a), f(b)], \tag{7}$$

$$\implies \int_a^b P_X(x)dx = \int_{f(a)}^{f(b)} P_Y(y)dy, \tag{8}$$

$$= \int_a^b P_Y(f(x))f'(x)dx. \tag{9}$$

As this must be true for every interval $[a,b]$, the integrands must be equal, giving us the relation

$$P_Y(y) = P_Y(f(x)) = \frac{P_X(x)}{|f'(x)|}. \tag{10}$$

where the absolute value appears in the denominator to ensure the probability distribution is positive everywhere. The point here is that, given a random variable X with a certain distribution, we can get a new random variable Y with a different distribution by applying a function to it. The idea behind the transformation method is to try find a function f which, when we compose it with a random variable, gives us a new random variable which is distributed the way we want.

If we can generate numbers distributed under P_X and we want random numbers distributed under P_Y, how can we find the appropriate function f to use? We can use (7) to write,

$$\int_{-\infty}^{x} P_X(\tilde{x})d\tilde{x} = \int_{-\infty}^{f(x)} P_Y(y)dy, \tag{11}$$

and by computing these integrals we may be able to find f. For instance, if the distribution P_X is the uniform distribution (5) and $x \in [0,1]$, then we can write

$$\int_{-\infty}^{x} P_X(\tilde{x})d\tilde{x} = \int_{-\infty}^{0} P_X(\tilde{x})d\tilde{x} + \int_0^x P_X(\tilde{x})d\tilde{x} \tag{12}$$

$$= \int_{-\infty}^{0} (0)d\tilde{x} + \int_0^x d\tilde{x} = x. \tag{13}$$

Hence, in this case (11) becomes

$$x = \int_{-\infty}^{f(x)} P_Y(y)dy. \tag{14}$$

Now, computing the integral on the right hand side of the above equation will give us x as a function of y. Inverting the resulting expression gives us the desired function f. As an example, we go back to the problem of generating points uniformly distributed on a sphere. Recall, this problem was solved by generating random numbers between 0 and π distributed according to $\frac{1}{2}\sin(\theta)$.

Example

We want to generate random numbers y with the following distribution:

$$P(y) = \frac{\sin(y)}{2}, \text{ where } 0 < y < \pi.$$

Substituting this distribution in for P_Y in (14) gives us the following:

$$\begin{aligned} x = \int_0^y \frac{\sin(\tilde{y})}{2} d\tilde{y} &= \left[-\frac{\cos(\tilde{y})}{2} \right]_0^y \\ &= \frac{1}{2}[1 - \cos(y)]. \end{aligned}$$

Inverting this expression yields

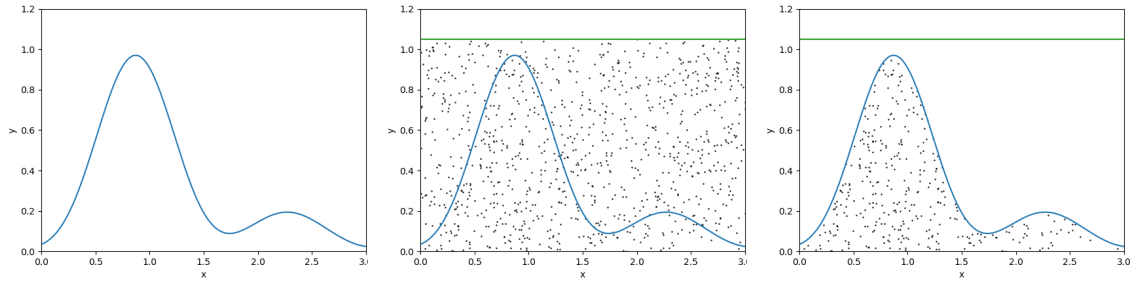
$$y = \cos^{-1}(1 - 2x).$$

Thus, generating a uniformly distributed number x between 0 and 1 and then plugging it into the above expression produces a random number y with the desired distribution. (Notice, this justifies the line we added to our python script to get a uniform distribution of points on a sphere.)

The transformation method, together with the prngs in numpy, gives us the ability to generate random numbers under a large class of distributions. However, there are situations where it's not possible to use the transformation method. For instance, the integral in (14) may not be expressible in terms of elementary functions, and even if it is there's no guarantee that the expression is invertible. For situations where we're unable to use the transformation method we can use the rejection method.

1.2.2 The Rejection Method

The rejection method (or the accept-reject algorithm) is a method for generating random numbers which are distributed according to *any* probability density function. The idea behind the rejection method can be illustrated by the following exercise: First, choose a probability density function $p(x)$ on the interval $[a, b]$ and plot the graph of $p(x)$ in the xy-plane (let's say the graph looks like Fig.3a). Then, draw a horizontal line, a height M above the x-axis, such that the graph of $p(x)$ is below this line and choose a number of random points uniformly distributed in the xy-plane above the interval $[a, b]$ and beneath our horizontal line as in Fig.3b. If we now throw away all of the randomly chosen points above the graph of $p(x)$ (like in Fig.3c), the x-coordinate of the remaining points will be distributed in the interval $[a, b]$ according to $p(x)$.



(a) (b) (c)

Figure 3

The key step, in the above exercise, for producing random numbers with distribution $p(x)$, is the throwing away of points between the graph of $p(x)$ and the horizontal line of height M. This step ensures the probability of a point remaining and not being thrown away is high where $p(x)$ is high and low where $p(x)$ is low. Notice, if the y-coordinates of the randomly chosen points are generated by first generating a uniform number u between 0 and 1 and then multiplying u by M (i.e. $y_i = Mu_i$), then we can rephrase this key step as follows: accept the randomly chosen point (x_i, y_i) if $u_i < p(x)/M$, otherwise throw it away.

What we've just described is called the rejection method for producing random numbers with distribution $p(x)$ from a set of random numbers uniformly distributed on the same domain (Producing random points under the straight line of height M required the x-coordinates of the points to be uniform in $[a, b]$). However, the use of a uniform distribution to begin with isn't necessary. Instead of drawing a straight line above the graph of $p(x)$, we could have drawn the graph of any other probability distribution $f(x)$ times a constant M such that the graph of $f(x)M$ is above the graph of $p(x)$ at every point x. Repeating the exercise with this new curve, instead of a straight line, leads us to the following recipe for generating numbers with distribution $p(x)$:

Suppose we're given generators for producing random numbers u distributed uniformly between 0 and 1 and X distributed according to $f(x)$. Given a constant $M \in \mathbb{R}$ such that $p(x) < Mf(x)$ for all $x \in \mathbb{R}$, the following three steps will produce a random number Y distributed according to $p(x)$:

1. Generate a random number X according to $f(x)$.
2. Generate a uniformly distributed random number u between 0 and 1.
3. If $u < p(X)/Mf(X)$, accept $Y = X$. Other wise reject X and execute these three steps again.

To prove Y is distributed according to $p(x)$, we first show that the probability of Y being less than x is given by

$$P(Y < x) = \int_{-\infty}^{x} p(\tilde{x})d\tilde{x}. \tag{15}$$

We note, for Y to be less than x, two things must be true. Firstly, the random number u must be less than $p(X)/Mf(X)$. Then, provided that's true, X must be less than x. Hence, we have

$$P(Y < x) = P(X < x | u < p(X)/Mf(X)), \tag{16}$$

$$= \frac{P(X < x, u < p(X)/Mf(X))}{P(u < p(X)/Mf(X))}. \tag{17}$$

We now note, that since X and u are independent random variables, the tuple (X, u) is distributed in the plane according to the product of distributions for X and u.

$$(X, u) \sim P(x, y) = f(x)P_{\text{uni}}^{[0,1]}(y). \tag{18}$$

Rewriting the probabilities appearing in (17) as integrals of the above distribution yields

$$P(Y < x) = \frac{\int_{-\infty}^{x} \left(\int_0^{p(\tilde{x})/Mf(\tilde{x})} f(\tilde{x}) dy \right) d\tilde{x}}{\int_{-\infty}^{+\infty} \left(\int_0^{p(\tilde{x})/Mf(\tilde{x})} f(\tilde{x}) dy \right) d\tilde{x}} \tag{19}$$

$$= \frac{\int_{-\infty}^{x} \left[p(\tilde{x})/Mf(\tilde{x}) \right] f(\tilde{x}) d\tilde{x}}{\int_{-\infty}^{+\infty} \left[p(\tilde{x})/Mf(\tilde{x}) \right] f(\tilde{x}) d\tilde{x}} \tag{20}$$

$$= \frac{\int_{-\infty}^{x} p(\tilde{x}) d\tilde{x}}{\int_{-\infty}^{+\infty} p(\tilde{x}) d\tilde{x}} \tag{21}$$

$$= \int_{-\infty}^{x} p(\tilde{x}) d\tilde{x} \tag{22}$$

The probability density of Y is given by the derivative of its cumulative distribution. So the distribution of Y must be

$$P_Y(y) = \frac{d}{dx} \left(\int_{-\infty}^{x} p(\tilde{x}) d\tilde{x} \right) \Big|_{x=y} = p(y), \tag{23}$$

proving that Y is distributed according to $p(x)$.

To give a hand waving explanation of the rejection method we first note the constant M is related to the number of rejections made while executing the algorithm. Specifically, if we want N numbers distributed under $p(x)$, we will typically need to generate about $M \times N$ numbers under $f(x)$ during execution (For this method to be efficient, we'd like M to be as close to 1 as possible). Now, if we generate $M \times N$ numbers distributed according to $f(x)$, we'll have about $f(x)dxM \times N$ numbers in the interval $(x, x + dx)$. Whereas we want N numbers under $p(x)$ such that we have about $p(x)dxN$ numbers in the interval $(x, x + dx)$. Therefore we need to throw away (or reject) $\frac{p(x)N}{f(x)M \times N}$ of the numbers we have in $(x, x + dx)$. We achieve this by only accepting numbers in $(x, x + dx)$, generated under $f(x)$, with probability $\frac{p(x)}{f(x)M}$.

1.3 Monte Carlo integration

Monte Carlo integration is a method for integrating a function over some domain, which utilises a theorem from probability theory known as the law of large numbers. The law of large numbers states that the mean of N independent, identically distributed random variables tends to the average value of the variables as N tends to infinity. In the context of integrating a function f over a domain A, this means if we choose N random points x_i in A, distributed according to $p(x)$, then the mean of the numbers $f(x_i)$ tends to the average of f with respect to $p(x)$ as N tends to infinity.

$$\sum_{i=1}^{N} \frac{f(x_i)}{N} \quad \xrightarrow[N\to\infty]{} \quad \langle f \rangle_p = \int_A f(x)p(x)dx. \tag{24}$$

To see how this helps us evaluate integrals, consider the integral of the function f over the set $A \subset \mathbb{R}^n$:

$$I = \int_A f(x)dx. \tag{25}$$

If we let X be a random variable taking values in $\mathbb{R}^n$ with probability

$$P_X(x) = \begin{cases} \dfrac{1}{\mathrm{Vol}(A)}, & \text{when } x \in A \\ 0 \text{ otherwise} \end{cases} \tag{26}$$

Then, the average value of $f(X)$ is

$$\langle f(X) \rangle = \int_{\mathbb{R}^n} f(x)P_X(x)dx = \int_A \frac{f(x)}{\mathrm{Vol}(A)}dx = \frac{I}{\mathrm{Vol}(A)}. \tag{27}$$

Hence the integral of f over A is equal to the volume of A times the average of f. Therefore, we can estimate the integral I by, firstly, generating a large number N of uniformly distributed points in A and then calculating the mean of f evaluated at the these points to estimate the average of f in A. Multiplying this estimate by the volume of A gives us an estimate for the value of I. The law of large numbers provides assurance our estimate of I tends to the true value of I as N grows. We denote our estimate of I by I_{MC}.

$$I = \mathrm{Vol}(A)\langle f \rangle \approx \frac{\mathrm{Vol}(A)}{N} \sum_{i=1}^{N} f(x_i) = I_{MC}. \tag{28}$$

Once we've estimated an integral using his technique, the natural thing to do is ask about how accurate our estimate is. We need some way of measuring how wrong our answer is. The way we measure the error is to think of our estimate I_{MC} as a random variable taking values in $\mathbb{R}$ (or whatever the codomain of f is). Our estimate of I is a function of N random variables uniformly distributed in A, so I_{MC} is itself also a random variable with its own distribution in $\mathbb{R}$. The distribution of I_{MC} will be, for large enough N, a sharply peaked bell shaped distribution centred around the

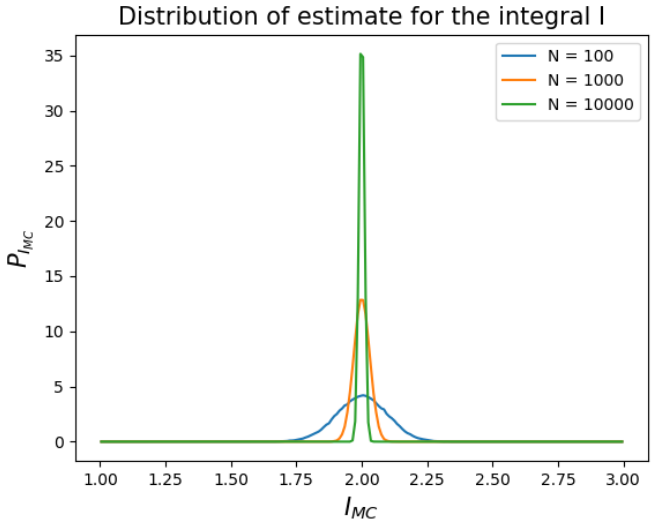


Figure 4: Here we take $I = \int_0^\pi \sin(x)dx = 2$ and plot the distribution of I_{MC} for different values of N. These curves were calculated by sampling I_{MC} 100,000 times for each N and using the function *numpy.histogram()* to get their distribution. We see for large N the distribution is sharply peaked around I.

true value of I (This is ensured by the central limit theorem) see Fig.4. The error of I_{MC} can be measured by measuring how spread out this distribution is around I. The standard way to measure how spread out a distribution is, is to calculate its standard deviation which is defined as the square root of the distributions variance.

$$\text{Err}(I_{MC}) = \sigma_{I_{MC}} = \sqrt{\text{Var}(I_{MC})}. \tag{29}$$

Recall, the variance of a random variable X is defined as the average square distance of X from the average of X ($\langle X \rangle$).

$$\begin{aligned} \text{Var}(X) &= \int P(x) \left[X^2 - \langle X \rangle^2\right] dx, & (30)\\ &= \int P(x)X^2 dx - \langle X \rangle^2 \int P(x)dx, & (31)\\ &= \langle X^2 \rangle - \langle X \rangle^2. & (32) \end{aligned}$$

We could calculate the error of I_{MC} by sampling I_{MC} a large number of times in order to calculate $\langle I_{MC}^2 \rangle$ and $\langle I_{MC} \rangle$, however this would be a lot of work. We find a better approach by subbing I_{MC} into the above formula for the variance of a random variable:

$$\begin{aligned}
\mathrm{var}(I_{\mathrm{MC}}) &= \langle (I_{\mathrm{MC}} - \langle I_{\mathrm{MC}} \rangle)^2 \rangle, && (33)\\
&= \langle (I_{\mathrm{MC}})^2 \rangle - \langle I_{\mathrm{MC}} \rangle^2, && (34)\\
&= \left\langle \left(\frac{V}{N} \sum_{i=1}^{N} f(x_i) \right)^2 \right\rangle - \left\langle \frac{V}{N} \sum_{i=1}^{N} f(x_i) \right\rangle^2, && (35)\\
&= \frac{V^2}{N^2} \left[\left\langle \left(\sum_{i=1}^{N} f(x_i) \right)^2 \right\rangle - \left\langle \sum_{i=1}^{N} f(x_i) \right\rangle^2 \right], && (36)\\
&= \frac{V^2}{N^2} \left[\left\langle \sum_{i=1}^{N} f(x_i)^2 + \sum_{\substack{i,j=1 \\ i \neq j}}^{N} f(x_i) f(x_j) \right\rangle - (N \langle f \rangle)^2 \right], && (37)\\
&= \frac{V^2}{N^2} \left[N \langle f^2 \rangle + \sum_{\substack{i,j=1 \\ i \neq j}}^{N} \langle f \rangle \langle f \rangle - N^2 \langle f \rangle^2 \right], && (38)\\
&= \frac{V^2}{N^2} \left[N \langle f^2 \rangle + \frac{N(N-1)}{N} \langle f \rangle^2 - N^2 \langle f \rangle^2 \right], && (39)\\
&= \frac{V^2}{N} \left[\langle f^2 \rangle - \langle f \rangle^2 \right]. && (40)
\end{aligned}$$

So when we're in the process of estimating $\langle f \rangle$ we should also estimate $\langle f^2 \rangle$ (which can be done at the same time). We can then calculate the error of I_{MC} as:

$$\mathrm{Err}(I_{MC}) = \frac{V}{\sqrt{N}} \sqrt{\langle f^2 \rangle - \langle f \rangle^2}. \tag{41}$$

It's worth noting at this point that the error of our estimate decreases like $\frac{1}{\sqrt{N}}$ as N increases. Moreover, the error is independent of the dimension d of the domain being integrated over. This is contrary to quadrature methods whose error grows as the dimension increases (e.g.. for the trapezium method Err $\approx N^{-2/d}$ which is better than that for Monte Carlo if $d < 4$). This increase in error due to an increase in dimension is sometimes referred to as the curse of dimensionality and is why Monte Carlo methods are preferred over quadrature methods in high dimensions.

1.3.1 Importance Sampling

Up until now, we wanted to generate points with a uniform distribution over a domain we're trying to integrate over in order to avoid bad estimates. However, there are situations where it is beneficial to generate points with a different distribution. These are situations where the function we're trying to integrate happens to be zero in a large part of the domain (or at least close to zero). The idea is to choose a distribution which is unlikely to generate points where f is zero. That way we're not wasting time

integrating zero as the only parts of the domain that matter in an integral are the parts where f is non-zero. Before seeing how this works in practice we note that the Monte Carlo method of integrating is based on the *law of large numbers* which states: if f is a function and x_i with $i = 1, 2 \cdots N$ are N random points distributed under $P(x)$, then in the limit $N \to \infty$:

$$\frac{1}{N}\sum_{i=1}^{N} f(x_i) \to \langle f \rangle_P \equiv \int_{-\infty}^{\infty} f(x)P(x)dx. \tag{42}$$

Notice, if $P(x) = \frac{1}{V}$ in a region of volume V and zero everywhere else (i.e. is uniform), we recover our method for estimating the average value of f over a the region.

$$\frac{1}{N}\sum_{i=1}^{N} f(x_i) \to \langle f \rangle \equiv \frac{1}{V}\int_{V} f(x)dx. \tag{43}$$

Now, to illustrate the idea, say we want to integrate f over the region V and we have a generator of random points distributed according to $q(x)$, then we can rephrase the integral as

$$I = \int_V f(x)dx = \int_V \frac{f(x)}{q(x)}q(x)dx. \tag{44}$$

Then, using the law of large numbers, we can approximate this integral by generating a large number N of random points x_i distributed according to $q(x)$, evaluating the quantity $\frac{f(x_i)}{q(x_i)}$ for each x_i and then calculating their average.

$$\frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i)}{q(x_i)} \approx I. \tag{45}$$

As mentioned, this method only really improves the accuracy of our calculation if the probability distribution $q(x)$ is close to zero when f is close to zero and non-zero where f is non-zero. More definitively, this method gives us an improvement when the fraction $\frac{f(x)}{q(x)}$ is roughly constant.

$$\text{If } \frac{f(x)}{q(x)} \approx \text{ constant, then } \sigma = \frac{1}{\sqrt{N}}\sqrt{\left\langle \frac{f^2}{q^2} \right\rangle - \left\langle \frac{f}{q} \right\rangle^2} \approx 0. \tag{46}$$

1.4 Stochastic processes

We mentioned that the Monte Carlo method is our preferred method of integration in high dimensions due to the error's independence of the dimension. However, generating random points with a particular distribution in high dimensions can be a lot of work using the methods discussed so far. For example, if we wanted to integrate over a domain whose dimension is 10'000, we would have to generate 10'000 numbers just to get a single random point in the domain. As we need a large number of points in the domain to do the integration accurately, this quickly turns into a lot of work.

A better way of doing this is to use what's called a stochastic process to simulate the distribution used in the integration. In this section, we'll discuss how this is done after introducing stochastic processes.

We can think of stochastic processes as the opposite of a deterministic process, which we're already familiar with. A system is deterministic if we're able to predict what the system will do at any time in the future, provided we know what the current state of the system is. These systems are usually modelled by differential equations. A stochastic process on the other hand, is a process where we can't predict what will happen, we can only say what will probably happen. Brownian motion is the classic example of a stochastic process but other examples include repeatedly flipping a coin (known as a Bernoulli process), electrical fluctuations in a circuit due to thermal noise and the occurrence earthquakes. Instead of differential equations, this type of process is modelled by a sequence of random variables. This brings us to the following definition:

Definition: A stochastic process is a sequence of random variables,

$$X_1, X_2, \cdots, X_i, \cdots,$$

all of which take values in the same set S and are distributed over S according to the following probability densities respectively:

$$P_1, P_2, \cdots, P_i, \cdots.$$

The set S is usually taken to be the configuration space or phase space of a physical system and so we refer to the set S as state space (e.g. for a particle in Brownian motion S is the set of possible positions of the particle). We think of the subscripts as time, i.e. X_1 is the state of the system at time 1 etc. For the duration of this section and for convenience, we will consider time to be discrete but it's possible to define stochastic processes with continuous time which are aptly called continuous-time stochastic processes. The probability densities P_i need not be the same for all i and can have complicated dependencies on the random variables X_i.

There's a special class of stochastic processes which we'll be interested in known as Markov chains. A Markov chain is a stochastic process which has the Markov property: *The distribution P_{i+1} only depends on i and the value taken by X_i.* For a process with this property, the distribution of the a particular X depends only on the previous X and is conditionally independent of all the X's that happened before the previous one. It is sometimes said a Markov process has *no memory* of its past as what happens next only depends on the current state.

Example (Markov Chain)

As an example of a Markov chain we consider a 1D random walk (or 1D Brownian motion). We imagine a particle beginning at a point x_0 on the real line and at every time i the particle moves to a new location on the line such that the displacement of the particle is a Gaussian random number ξ. To realise this process in python, we can implement the following steps for some chosen values of σ and x_0:

1. Generate a number $\xi \sim \text{Norm}(0, \sigma)$ and set $x_1 = x_0 + \xi$.
2. Generate a number $\xi \sim \text{Norm}(0, \sigma)$ and set $x_2 = x_1 + \xi$.

$\vdots$

i. Generate a number $\xi \sim \text{Norm}(0, \sigma)$ and set $x_i = x_{i-1} + \xi$.

$\vdots$

This process should continued until we reach some desired number of steps. If $G \sim \text{Norm}(0, \sigma)$ is a Gaussian random variable, then the sequence of random variables describing this process is:

$$\begin{aligned} X_1 &= x_0 + G, \quad (\sim P_1 = \text{Norm}(x_0, \sigma)\ ) \\ X_2 &= x_1 + G, \quad (\sim P_1 = \text{Norm}(x_1, \sigma)\ ) \\ &\vdots \\ X_i &= x_{i-1} + G, \quad (\sim P_1 = \text{Norm}(x_{i-1}, \sigma)\ ) \end{aligned}$$

This is clearly a Markov chain since each P_i depends on the outcome of X_{i-1}.

The following code implements the Markov chain (or random walker) described in the above example for 100 steps. It then plots the outcome of random variables X_i as a function of i (or the history of the walker). This code was ran three times to produce the graph show in Fig. 5.

Figure 5: The histories of three random walkers

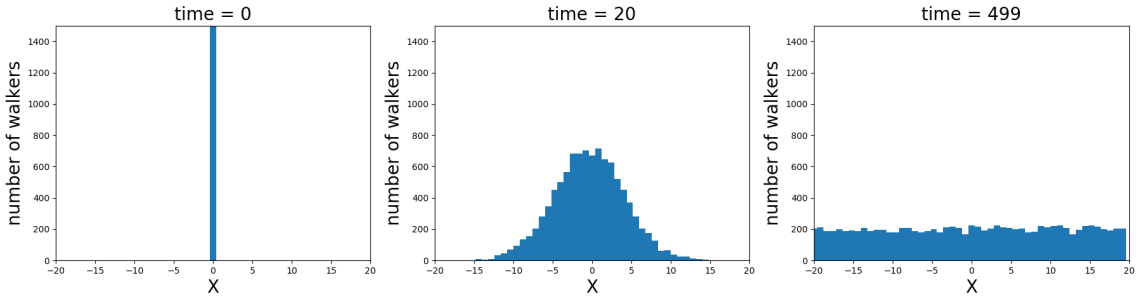
```
import numpy as np
import numpy.random as ran
import matplotlib.pyplot as plt

# number of time steps
N_steps = 100

# allocate memory to store the history of the walker
x = np.zeros(N_steps)

# every time step, move the walker by a Gaussian random number
for i in np.arange(1,N_steps):
    x[i] = x[i-1] + ran.randn()

# plot the history of the walker
plt.plot(np.arange(0,N_steps),x)
plt.xlabel('time')
plt.ylabel('x')
plt.title('History of random walker')
```

Let's now consider a large number of these walkers evolving in time simultaneously and how their distribution on the real line changes in time. The following code creates ten thousand random walkers and evolves them under a similar Markov process to the one described above. The only difference is that now the process reflects the walkers off two boundary points $x_{\max}$ and $x_{\min}$ so that the walkers are now confined to the 1 dimensional box $[x_{\max}, x_{\min}]$. The code also animates the distribution of the walkers as they evolve under this Markov process. Initially, all ten thousand walkers are located the origin and we see the distribution peaked sharply at that point. Then, as the walkers start moving, a bunch of walkers drift to the left, another bunch drift to the right and some fluctuate about the origin resulting in the distribution spreading

out until it fills the box. Eventually, the distribution settles to a uniform distribution as shown in Fig.6.

```
import numpy as np
import numpy.random as ran
import matplotlib.pyplot as plt
import matplotlib.animation as ani

def next_position(x, xmax):
    ''' This function updates the position of a
        collection of walkers'''

    N = len(x)
    x = x + ran.randn(N)          # move all the walkers
                                  # by a Gaussian

    I = np.where(x>xmax)          # Implement boundary at
    x[I] = xmax - (x[I]-xmax)     # x = xmax

    I = np.where(x<-xmax)         # Implement boundary at
    x[I] = -xmax + (-xmax-x[I])   # x = -xmax
    return x + ran.randn(N)

# Number of walkers and initial position of walkers
walkers = 10**4; x0 = 0.0

# Number of bins for histogram and boundary positions
bins = 50; xmax = 20; xmin = -1*xmax

# Number of time steps & allocate memory
nts = 500
x = x0 + np.zeros(walkers)        # positions of walkers
counts = np.zeros((nts, bins))    # distribution of walkers
counts[0,:], edges = np.histogram(x, bins=bins,
                                  range=(xmin, xmax))

# Evolve walkers & calculate their distribution each time step
for i in range(1, nts):
    x = next_position(x, xmax)
    counts[i, :],_ = np.histogram(x, bins=bins,
                                  range=(xmin, xmax))

# This function tells FuncAnimation what to draw each time step
def update_hist(num, counts, edges, xlims, ylims):
    plt.cla()
    plt.bar(edges[:-1], counts[num,:])
    plt.xlim(xlims)
    plt.ylim(ylims)
```

```
# This animates the distribution of walkers
fig = plt.figure()
movie = ani.FuncAnimation(fig, update_hist, nts,
                          fargs=(counts,edges, (xmin, xmax),
                          (0, 1000)))
plt.show()
```



(a) (b) (c)

Figure 6

The fact that the distribution of walkers eventually settles down on a uniform distribution is a property of this particular Markov chain. If we were to update the walkers using something other than a Gaussian random number, we would have a different Markov chain and the distribution of walkers may settle down on a different distribution. To illustrate this, and to introduce useful notation, we'll alter the Markov chain implemented above by including what we'll call an acceptance step. The general form of a Markov chain with an acceptance step is expressed in the following 4 steps:

1. Start the process in a predefined state μ_0
2. (Trial step) Choose a trial state ν for the next state of the process. The probability of choosing state ν given the process is in state μ is denoted $\omega_{\mu\nu}$.
3. (Acceptance step) Accept ν as the next state with probability $A_{\mu\nu}$. Otherwise, leave the process where it is by accepting μ as the next state instead.
4. Return to step 2.

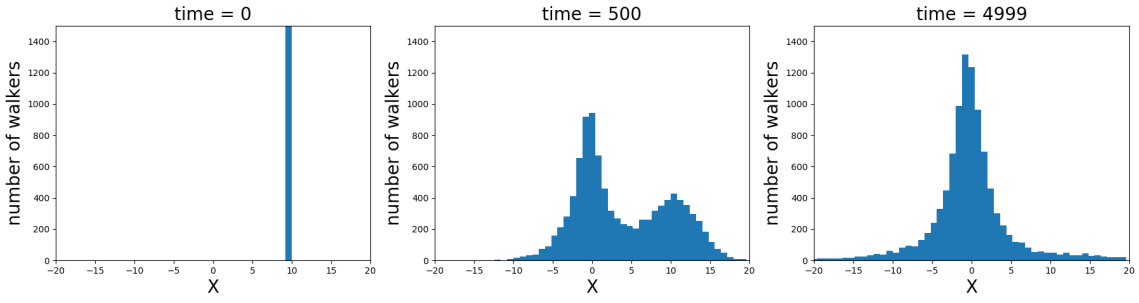
Let's now take the Markov chain we previously considered and make the following change to each step: after generating a Gaussian random number ξ, we accept $x_{i+1} = x_i + \xi$ as the next state of the process with probability

$$A = \frac{1}{1 + 0.25(x_i + \xi)^2}. \tag{47}$$

Otherwise we set $x_{i+1} = x_i$. This can be implemented by replacing the *next_position()* function in the python code with the following python function.

```
def next_position(x, xmax):
    ''' This function updates the position of a
        collection of walkers'''
    N = len(x)

    # Trial step
    xi = ran.randn(N)

    # Acceptance step
    for i in range( N ):

        # For each walker, accept the new state
        # with probability A_{mu nu} :
        if ran.rand()< 1.0/( 1 + 0.25*( x[i] + xi[i] )**2 ):
            x[i] = x[i] + xi[i]

    # return the new states of the walkers
    return x
```

With this alteration to the code, walkers are less likely to move away from the origin the further away they are from the origin. If we now run the code and, for example, set the starting point $x_0 = 10$, initially we see a sharp peak in the distribution of walkers at $x = 10$. Then, as before, the distribution starts to spread out as walkers begin evolving. However, instead of the distribution spreading out to fill the box, the walkers are now pushed towards the origin and their distribution eventually comes to rest where it well approximates a Cauchy distribution centred at the origin as shown in Fig. 7.



(a) (b) (c)

Figure 7

Consequently, if we wanted to generate some random numbers with a Cauchy distribution, we could implement the Markov chain above and once the distribution of walkers reaches equilibrium, we could take the positions of the walkers as our desired random numbers. This suggests yet another way of sampling points from a distribution which is very different from the transformation and rejection methods. If we want to sample from the distribution $P(x)$, the idea is to find a Markov process which will

evolve a collection of walkers in such a way that the distribution of walkers converges to $P(x)$ where it comes to equilibrium. Once the distribution is in equilibrium, we can stop the evolution and accept the current state of each walker as a sample from the distribution $P(x)$. In practise we only evolve one or a small collection of walkers and periodically accept their states as samples once they've reached equilibrium. The equilibrium distribution of a Markov process is also called a Gibbs state.

The natural question to ask now is, if we're interested in a particular distribution $P(x)$, how do we design a Markov process which has $P(x)$ as its equilibrium distribution. To answer this question we need to know how a distribution changes in time as the system evolves under a particular Markov chain. To this end, we now introduce transition probabilities and use them to write down what's called a master equation for the system. For the rest of this section, to simplify notation, we'll assume the state space S of every Markov chain is discrete and states can be indexed by $\mathbb{N}$ or some finite subset of $\mathbb{N}$.

For any Markov chain, a transition probability $T_{\nu\mu}$ tells us the probability the next state the system will be in is ν given that the system is currently in state μ.

$$\Pr(\text{ Next state is } \nu \mid \text{Current state is } \mu) = T(\mu \to \nu) = T_{\nu\mu}. \tag{48}$$

In general the transition probabilities of a Markov chain can depend on time but for the Markov chains we'll be considering here, the transition probabilities will be independent of time. Note, since at any given time during a Markov chain the system must be in a state contained in S, the transition probabilities must satisfy $\sum_{\nu \in S} T_{\nu\mu} = 1$. If a Markov chain fits the general form of a Markov chain with an acceptance step described above, the transition probabilities can be written as $T_{\nu\mu} = \omega_{\mu\nu} A_{\mu\nu}$.

At any time i the distribution of a collection of walkers is given by the following probability distribution

$$P_\mu(i) = \frac{\text{Number of walkers in state } \mu \text{ at time } i}{\text{Total number of walkers}}. \tag{49}$$

In the special case where the total number of walkers is infinite, we can use the distribution at time i and the transition probabilities to calculate the distribution at time $i+1$. This is done using the fact that the number of walkers in state ν at time $i+1$ is the sum over S of the number of walkers at time i transitioning from μ to ν. Heuristically, we can express this fact as follows:

$$\begin{pmatrix} \text{Number of walkers in} \\ \text{state } \nu \text{ at time } i+1 \end{pmatrix} = \sum_{\mu \in S} \begin{pmatrix} \text{Fraction of walkers in state} \\ \mu \text{ that move to state } \nu \end{pmatrix} \begin{pmatrix} \text{Number of walkers} \\ \text{in state } \mu \text{ at time } i \end{pmatrix}$$

We now divide both sides of this expression by the total number of walkers and note that as the total number of walkers grows, the fraction of walkers transitioning from μ to ν better approximates $T_{\nu\mu}$. Thus, in the limit where the number of walkers goes to infinity, we can write

$$P_\nu(i+1) = \sum_{\mu \in S} T_{\nu\mu} P_\mu(i). \tag{50}$$

This equation is sometimes written using matrix notation as $P_{i+1} = TP_i$ where P_i is a vector containing the probabilities $P_\mu(i)$ and T is a matrix containing the transition probabilities $T_{\nu\mu}$. The condition $\sum_{\nu\in S} T_{\nu\mu} = 1$ implies the elements of any column of the matrix T must add up to one. Such a matrix is called a stochastic matrix.

By subtracting $P_\nu(i)$ from the expression for $P_\nu(i+1)$ and using the relation $\sum_{\nu\in S} T_{\mu\nu} = 1$ we come to the following expression describing how the distribution of an infinite collection of walkers evolves:

$$P_\nu(i+1) - P_\nu(i) = \sum_{\mu\in S} T_{\nu\mu}P_\mu(i) - P_\nu(i)\left(\sum_{\mu\in S} T_{\mu\nu}\right) \tag{51}$$

$$= \sum_{\mu\in S} \left(T_{\nu\mu}P_\mu(i) - P_\nu(i)T_{\mu\nu}\right). \tag{52}$$

This is called the master equation for the Markov chain and relates the transition probabilities of the Markov chain to how the distribution of an infinite collection of walkers changes in time. For a finite collection of walkers we would expect their distribution to evolve according to (52) on average, rather than exactly.

We're ultimately interested in creating a Markov chain which has a desired distribution $\tilde{P}_\mu$ as an equilibrium distribution. As the equilibrium distribution doesn't change in time we drop the dependence on time the variable i in the notation. To ensure the distribution $\tilde{P}_\mu$ is stationary under our Markov chain (so the left hand side of the master equation is zero), we can require that each term in the sum on the right hand side of the master equation is zero. This gives the following condition known as detailed balance which the transition probabilities of our Markov chain must satisfy in order to have the desired result:

$$\tilde{P}_\mu T_{\nu\mu} - \tilde{P}_\nu T_{\mu\nu} = 0, \quad \text{or} \quad \tilde{P}_\mu T_{\nu\mu} = \tilde{P}_\nu T_{\mu\nu}. \tag{53}$$

While the condition of detailed balance ensures our desired distribution $\tilde{P}$ is stationary under our Markov chain, it alone is not enough to ensure the distribution of walkers converges to $\tilde{P}$. If we being with an arbitrary distribution P, it's possible our Markov chain may evolve P to some stationary distribution other than the one we wanted. To guarantee that we end up with the desired distribution, regardless of what initial distribution we start with, we can require the Markov chain to satisfy the additional condition of *ergodicity*. A Markov chain is ergodic if it is possible for a walker to reach any state in S from any other state in a finite number of steps. The reason why the conditions of detailed balance and ergodicity ensure the Markov chain converges to the desired distribution is left as an aside in the section 1.4.1. A method for creating Markov chains satisfying detailed balance and ergodicity is then discussed in section 1.4.2.

1.4.1 (Aside) Convergence of Markov chains

The proof that a Markov chain satisfying detailed balance and ergodicity will converge to the desired distribution $\tilde{P}$, regardless of the initial distribution we choose, uses three

results involving the matrix of transition probabilities T. We derive each result in turn and then use them to prove that the Markov chain will converge as stated. The three results are as follows:

1. The matrix T being stochastic and non-negative (meaning the entries of T are greater than or equal to zero) implies every eigenvalue λ of T satisfies $|\lambda| \leq 1$.
2. Detailed balance and T being a stochastic matrix implies the vector $\tilde{P}$ is an eigenvector of T with eigenvalue 1.
3. Ergodicity implies any eigenvector of the stochastic, non-negative matrix T with eigenvalue 1 is unique.

To show the first result, recall that the entries of the matrix T are probabilities and so T is non-negative. Also, recall that T being a stochastic matrix means $\sum_\nu T_{\nu\mu} = 1$. Now, suppose v is a left eigenvector of T with eigenvalue λ. Without loss of generality, let the m-th component of v, i.e. v_m, be one of the components with the largest absolute value. So we have $|v_m| \geq |v_i|$ for every component v_i of v. By taking the absolute value of the m-th column of the eigenvalue equation $vT = \lambda v$ we find the following:

$$|\lambda v_m| = |\lambda||v_m| = |\sum_i T_{im} v_i| \tag{54}$$

$$\leq \sum_i T_{im}|v_i| \qquad \left(\begin{array}{l}\text{By the triangle inequality}\\ \text{and non-negativity of } T\end{array}\right) \tag{55}$$

$$\leq \sum_i T_{im}|v_m| \qquad \left(\begin{array}{l}\text{Since } v_m \text{ has the largest}\\ \text{absolute value}\end{array}\right) \tag{56}$$

$$= |v_m|. \qquad (\text{Since } T \text{ is stochastic}) \tag{57}$$

Thus $|\lambda||v_m| \leq |v_m|$ implying $|\lambda| \leq 1$ as required.

The second result follows from expressing the ν-th component of the vector $T\tilde{P}$ in summation notation and then using the detailed balance condition (53) to swap the indices being summed over.

$$(T\tilde{P})_\nu = \sum_\mu T_{\nu\mu}\tilde{P}_\mu = \sum_\mu \tilde{P}_\nu T_{\mu\nu} = \tilde{P}_\nu \sum_\mu T_{\mu\nu} = \tilde{P}_\nu. \tag{58}$$

In the last equality, we used the fact that T is stochastic. Hence, $\tilde{P}$ is an eigenvector of T with eigenvalue 1.

Before proving the third result we will show that ergodicity implies any eigenvector of T with eigenvalue 1 can be chosen so that all of its components are positive. The third result is a corollary to this. To start, recall ergodicity means any state ν can be reached from any state μ in a finite number of steps. In other words, for any two states μ and ν, there exists a finite positive integer $n_{\mu\nu}$ such that the μ, ν component of the matrix $T^{n_{\mu\nu}}$ is non-zero: $T^{n_{\mu\nu}}_{\nu\mu} > 0$. Let n be the largest $n_{\nu\mu}$ and define the matrix M as follows:

$$n = \max_{\mu,\nu\in S}\{n_{\mu\nu}\}, \quad M = \frac{1}{n}\left(T + T^2 + \cdots + T^n\right). \tag{59}$$

It's easy to see that M is also stochastic and any eigenvector of T is also an eigenvector of M. Moreover, since each matrix T^m is non-negative and because of the way n is defined, we have $M_{\nu\mu} > 0$ for all $\mu, \nu \in S$. We denote the smallest component of M by δ.

We now let v be an eigenvector of T with eigenvalue 1 and separate v into positive and negative parts. Namely, we define v^+ to be the vector whose components are the same as v after every negative component of v has been set to zero. Similarly, we define v^- to be the vector whose components are the same as $-v$ after every negative component has been set to zero. So we have $v = v^+ - v^-$. Also, let $\alpha = \min\{\sum_i v_i^+, \sum_i v_i^-\}$. Notice, each component of the vector Mv^+ is greater than or equal to $\delta\alpha$.

$$\sum_j M_{ij} v_j^+ \geq \sum_j \delta v_j^+ = \delta \sum_j v_j^+ \geq \delta\alpha. \tag{60}$$

Similarly, we have $\sum_j M_{ij} v_j^- \geq \delta\alpha$. We can now show that α must be zero by summing up the absolute value of the components of the vector Mv:

$$\begin{aligned}
\sum_i |(Mv)_i| &= \sum_{ij} |M_{ij} v_j|, \\
&= \sum_{ij} |M_{ij} v_j^+ - M_{ij} v_j^-|, \\
&\leq \sum_{ij} |M_{ij} v_j^+ - \delta\alpha| + |M_{ij} v_j^- - \delta\alpha|, && \text{(By the triangle inequality)} \\
&= \sum_{ij} |M_{ij} v_j^+| + |M_{ij} v_j^-| - 2\delta\alpha, && \text{(By (60))} \\
&= \sum_j |v_j^+| + |v_j^-| - 2N\delta\alpha, && \left(\begin{array}{l}\text{Since } M \text{ is stochastic,} \\ N = \text{total number of states}\end{array}\right) \\
&= \sum_j |v_j| - 2N\delta\alpha. && \big(\text{Since } v = v^+ - v^-\big)
\end{aligned}$$

However, $Mv = v$ and so $\sum_i |(Mv)_i| = \sum_i |v_i|$. Also, $\delta, N > 0$ so we must have $\alpha = 0$. Thus, either all the components of v are positive or they're all negative, in which case $-v$ has positive components.

Now, to see that any eigenvector v of T with eigenvalue 1 is unique we suppose that both v and u are such eigenvectors. We choose v and u so that all of their components are positive. We also choose to normalise v and u such that their components sum to 1: $\sum_i v_i = \sum_i u_i = 1$. Then the vector $v-u$ is also an eigenvector of T with eigenvalue 1 and so its components are also all positive or negative. However, $\sum_i (v_i - u_i) = \sum_i v_i - \sum_i u_i = 1 - 1 = 0$. Therefore, each component of $v-u$ must be zero, implying $v = u$.

We're now in a position to show that a Markov chain satisfying detailed balance and ergodicity will converge to the desired distribution. Assuming (for the moment) that the matrix T is diagonalisable, we can expand any initial distribution we choose as a combination of eigenvectors of T:

$$P = c_0 v_0 + c_1 v_1 + c_2 v_2 + \cdots, \tag{61}$$

where the v_i's are eigenvectors of T and the c_i's are coefficients. Since T satisfies detailed balance and ergodicity, we know our desired distribution $\tilde{P}$ is the only eigenvector of T with eigenvalue 1. We choose to let $v_0 = \tilde{P}$ and it can be shown that the corresponding coefficient c_0 must equal one (otherwise the distribution will not be normalised throughout the evolution of the Markov chain). So now we can write

$$P = \tilde{P} + c_1 v_1 + c_2 v_2 + \cdots . \tag{62}$$

After n steps of the Markov chain our initial distribution turns into

$$T^n P = T^n \tilde{P} + c_1 T^n v_1 + c_2 T^n v_2 + \cdots , \tag{63}$$

$$= \tilde{P} + c_1 \lambda^n v_1 + c_2 \lambda^n v_2 + \cdots . \tag{64}$$

However, we know that the eigenvalues, other than 1, satisfy $|\lambda_i| < 1$. Hence, in the limit $n \to \infty$, the v_i terms vanish and we have $T^n P \to \tilde{P}$. The same result is true if the matrix T is not diagonalisable. In this case we can consider T in a basis of generalised eigenvectors so that T is in Jordan normal form:

$$T = \begin{bmatrix} 1 & & & \\ & J_1 & & \\ & & \ddots & \\ & & & J_k \end{bmatrix}, \tag{65}$$

where the J_i's are Jordan blocks. After a large number of steps in the Markov chain, each Jordan block becomes:

$$J_i^n = \begin{bmatrix} \lambda_i & 1 & & \\ & \lambda_i & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_i \end{bmatrix}^n = \begin{bmatrix} \lambda_i^n & n\lambda_i^{n-1} & \cdots & \binom{n}{N-1}\lambda_i^{n-N} \\ & \lambda_i^n & \ddots & \\ & & \ddots & n\lambda_i^{n-1} \\ & & & \lambda_i^n \end{bmatrix}. \tag{66}$$

Since $|\lambda_i| < 1$ implies $\binom{n}{N-1}\lambda_i^{n-N} \to 0$ as $n \to \infty$, each J_i tends to the zero matrix. As $\tilde{P}$ is the eigenvector corresponding to the eigenvalue 1 in (65), this shows that $T^n P \to \tilde{P}$.

1.4.2 The Metropolis Algorithm

The Metropolis algorithm is a method for creating Markov chains that satisfy the condition of detailed balance. Let us continue to denote our desired distribution by $\tilde{P}$. The key idea is to choose a Markov chain with an acceptance step using the following acceptance probability.

$$A_{\mu\nu} = \begin{cases} 1 \text{ if } \tilde{P}_\mu \leq \tilde{P}_\nu, \\ \frac{\tilde{P}_\nu}{\tilde{P}_\mu} \text{ if } \tilde{P}_\mu > \tilde{P}_\nu. \end{cases} \tag{67}$$

We also need to ensure that the trial step of the Markov chain makes the process ergodic and is symmetric, meaning $\omega_{\mu\nu} = \omega_{\nu\mu}$. To see this Markov chain satisfies the condition of detailed balance we can look at the following quotient:

$$\frac{T_{\nu\mu}}{T_{\mu\nu}} = \frac{\omega_{\mu\nu}A_{\mu\nu}}{\omega_{\nu\mu}A_{\nu\mu}} = \frac{A_{\mu\nu}}{A_{\nu\mu}} = \frac{\tilde{P}_\nu}{\tilde{P}_\mu}. \tag{68}$$

This is the detailed balance condition (53).

One major advantage of this method is we don't need to be able to compute $\tilde{P}$ exactly, only up to an overall constant. This is very useful because the normalisation of a distribution can be difficult/expensive to calculate. For example, a common calculation where this method is particularly useful is calculating the average of an observable A of a system in thermal equilibrium at temperature T:

$$\langle A \rangle = \sum_{\mu \in S} A_\mu \frac{e^{-H_\mu/kT}}{Z}, \tag{69}$$

where A_μ is the value of A when the system is in state μ, H_μ is the energy of state μ, k is the Boltzmann constant and Z is the partition function. The issue in calculating this quantity directly is the state space S is often way too large to loop through. For a 20×20 Ising model (introduced in class) the state space has $2^{20\times 20} \approx 10^{120}$ states, so even if we had an extremely powerful computer that could compute 10^{15} of the terms in (69) per second, it would still take about 10^{97} years to compute them all. The trick is to notice that the Boltzmann distribution $P_\mu = \exp(-H_\mu/kT)/Z$ is small when H_μ is large. So we can compute a good approximation to (69) by using Monte carlo integration with importance sampling:

$$\langle A \rangle \approx \frac{1}{N} \sum_{i=1}^{N} A_{\mu_i}, \tag{70}$$

where the μ_i are distributed in S according to the Boltzmann distribution. We could implement a Markov chain to generate samples from the Boltzmann distribution but, depending on how we implement it, we may need to calculate the partition function Z to ensure the Markov chain satisfies detailed balance (53). However, computing Z is often an intractable problem. We can employ the Metropolis algorithm which avoids this problem since the acceptance probability (67) is what ensures detailed balance and in this case the acceptance probability becomes:

$$A_{\mu\nu} = \begin{cases} 1 & \text{if } H_\nu \leq H_\mu, \\ \exp(\frac{H_\mu - H_\nu}{kT}) & \text{if } H_\nu > H_\mu, \end{cases} \tag{71}$$

which is independent of the partition function Z.

1.4.3 The Heatbath Algorithm

(To be completed)

1.4.4 Autocorrelation

(To be completed)

2 Part B: Differential Equations

2.1 Matrices and Linear differential equations

Consider the following system of linear equations.

$$\begin{aligned} a_{11}w + a_{12}x + a_{13}y + a_{14}z &= b_1 \\ a_{21}w + a_{22}x + a_{23}y + a_{24}z &= b_2 \\ a_{31}w + a_{32}x + a_{33}y + a_{34}z &= b_3 \\ a_{41}w + a_{42}x + a_{43}y + a_{44}z &= b_4 \end{aligned} \tag{72}$$

Recall, such a system can be written as a matrix equation, of the form $\mathbf{Ax} = \mathbf{b}$, by collecting all the unknowns into a vector $\mathbf{x}$, all their coefficients into a matrix $\mathbf{A}$ and constant terms into a vector $\mathbf{b}$ as follows:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \tag{73}$$

Solving the system of linear equations then involves finding the inverse of the matrix $\mathbf{A}$ and calculating $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. Such a system of linear equations arise when we want to solve linear differential equations on a discretised space.

Consider the following linear ODE on the interval $[a, b]$:

$$A\frac{d^2\phi}{dx^2} + B\frac{d\phi}{dx} + C\phi = \rho, \quad \phi(a) = c_1, \quad \phi(b) = c_2. \tag{74}$$

We now discretise the defined interval into $(N + 2)$ points, with lattice spacing $dx = \frac{b-a}{N+1}$. An unknown function $\phi(x)$ defined on the original interval now becomes a discrete set of unknowns, one for each point of the lattice. We label the points of the lattice as $x = x_0 + ia$ and the value of the function ϕ at the lattice points as follows:

$$\phi(x) = \phi(x_0 + idx) = \phi_i, \tag{75}$$

where $i = 0, 1, \cdots N + 1$. Recall, on a discretised space, derivatives of a function are given by finite differences. We'll use the symmetric finite difference equations for both the first and second derivatives of a function in (74). In other words, we replace the derivatives in the differential equation (74) by the following:

$$\frac{\partial\phi}{\partial x}(x) \to \frac{\phi_{i+1} - \phi_{i-1}}{2dx}, \tag{76}$$

$$\frac{\partial^2\phi}{\partial x^2}(x) \to \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{dx^2}. \tag{77}$$

This yields

$$A\left(\frac{\phi_{i+1}-2\phi_i+\phi_{i-1}}{dx^2}\right)+B\left(\frac{\phi_{i+1}-\phi_{i-1}}{2dx}\right)+C\phi_i=\rho_i \tag{78}$$

Hence the differential equation (74) turns into the following set of N equations involving the unknowns ϕ_i:

$$\left(Cdx^2-2A\right)\phi_i+\left(A-\frac{Bdx}{2}\right)\phi_{i-1}+\left(A+\frac{Bdx}{2}\right)\phi_{i+1}=dx^2\rho_i, \tag{79}$$

where $i=1\cdots N$.

Note, not all of the ϕ_i's are unknown. Both ϕ_0 and ϕ_{N+1} are fixed by the boundary conditions. This means that not all of the above equations have a left hand side of the same form as that of the equations listed in (72). Namely the equations where $i=1$ and $i=N$ have constant terms on the left hand side. By bringing these terms to the right hand side, all of the above equations will have the same form as those in (72) and so we can write the system of equations as the following matrix equation.

$$\begin{bmatrix}\gamma & \beta & 0 & 0 & \cdots & 0\\ \alpha & \gamma & \beta & 0 & \cdots & 0\\ 0 & \alpha & \gamma & \beta & \cdots & 0\\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots\\ 0 & 0 & \cdots & \alpha & \gamma & \beta\\ 0 & 0 & \cdots & 0 & \alpha & \gamma\end{bmatrix}\begin{bmatrix}\phi_1\\ \phi_2\\ \vdots\\ \phi_{N-1}\\ \phi_N\end{bmatrix}=\begin{bmatrix}dx^2\rho_1-\alpha\phi_0\\ dx^2\rho_2\\ \vdots\\ dx^2\rho_{N-1}\\ dx^2\rho_N-\beta\phi_{N+1}\end{bmatrix} \tag{80}$$

where $\alpha=A-\frac{Bdx}{2}$, $\gamma=Cdx^2-2A$ and $\beta=A+\frac{Bdx}{2}$.

2.2 Boundary value PDEs

(To be completed)

2.3 Initial value PDEs

We now discuss solving linear initial value problems. As a running example we'll consider the diffusion equation in $1+1$ dimensions with initial value $u(x,0)=u_0(x)$,

$$\frac{\partial u}{\partial t}=D\frac{\partial^2 u}{\partial x^2}. \tag{81}$$

The first thing to do is discretise both the space and time dimensions into a rectangular grid. We do this in almost the same way we discretised the xy-plane. The difference is we use different lattice spacings in the x and t directions. The main issue with initial value problems and time evolution is the stability of our solution. While evolving a function u in time, in practise it can happen that errors accumulate in such a way that our answer veers off its true course and converges to an incorrect answer. To ensure stability it's sometimes essential for the spacing in the space direction to be

different to the spacing in the time direction and for both to satisfy an inequality that guarantees stability (which we'll soon derive). We denote the spacing in the space direction by dx and the spacing in the time direction by dt. Then, for the values of the function u at different sites of the space-time grid we write

$$u(x_i, t_n) = u(x_0 + idx, t_0 + ndt) \equiv u_i^n. \tag{82}$$

The next step is to replace the derivatives in the diffusion equation by finite difference equations. Up until now we've been using the symmetric finite difference equations for everything. We'll continue to do so for the spacial derivatives. However, it turns out that it's a bad idea to use the symmetric finite difference equation for the time derivative (we'll see soon that this leads to instabilities). Instead, we're going to first look at using the forward derivative for the time direction. Replacing the derivatives in the diffusion equation by the finite difference equations just discussed results in the following set of equations:

$$u_i^{n+1} = \left(1 - \frac{2Ddt}{dx^2}\right) u_i^n + \frac{2Ddt}{dx^2}(u_{i-1}^n + u_{i+1}^n). \tag{83}$$

We now have a matrix equation of the form $u^{n+1} = Au^n$ which relates the function u at time $n+1$ with u at time n. We can now solve the initial value problem by repeatedly acting on the initial value u_0 with the matrix A until we obtain u^n.

Now that we have one way of solving initial value problems, we can ask how stable it is. The standard way to check the stability of a method like the FTCS scheme is called Von Neumann analysis. The first step of the analysis is to replace the components of u in the finite difference equation by the components of its Fourier transform in the space direction: u_k^n. Each Fourier mode evolves independently in time for the diffusion equation. This gives the eigenmode evolution:

$$u_k^{n+1} = \xi_k u_k^n \implies u_k^n = \xi_k^n u_k^0. \tag{84}$$

So every time we multiply our solution by A to evolve it forward another step, we multiply each mode u_k^n by a constant ξ_k. We can find an expression for the constant ξ_k by inserting this into the FTCS scheme for the diffusion equation gives

$$\frac{\xi_k^{n+1} e^{ijkdx} - \xi_k^n e^{ijkdx}}{dt} = D\xi_k^n \frac{e^{i(j-1)kdx} - 2e^{ijkdx} + e^{i(j+1)kdx}}{(dx)^2}. \tag{85}$$

Letting $n = 0$ and rearranging gives:

$$\xi_k = 1 - \frac{2Ddt}{(dx)^2}\left(e^{-ikdx} - 2 + e^{ikdx}\right) = 1 - \frac{4Ddt}{(dx)^2}\sin^2\left(\frac{kdx}{2}\right). \tag{86}$$

For stability we need $|\xi_k| \leq 1$ for all k. However, the above equation is always less than one, so we need $\xi_k \geq -1$ for all k. Therefore, we need

$$\frac{4Ddt}{(dx)^2}\sin^2\left(\frac{kdx}{2}\right) \leq 2. \tag{87}$$

The worst case is when k is such that $\sin^2(\frac{k\delta x}{2}) = 1$. Hence, to ensure stability, we should choose

$$\frac{4Ddt}{(dx)^2} \leq 2 \implies dt \leq \frac{(dx)^2}{2D}. \tag{88}$$

CTCS scheme: (To be completed)(unconditionally unstable)
BTCS scheme: (To be completed)(unconditionally stable)
Crank-Nicolson method: (To be completed)(unconditionally stable)

2.4 Relaxation methods

The relaxation method for solving an equation like $\mathcal{L}\phi = \rho$, where $\mathcal{L}$ is an elliptic operator, ϕ is an unknown function and ρ is a known function, for some specified boundary values, involves turning the boundary value problem into an initial value problem. Basically, we choose an arbitrary initial state f and evolve it according to the following PDE:

$$\frac{\partial f}{\partial t} = \mathcal{L}f - \rho. \tag{89}$$

The expectation is that the system will eventually converge to a steady state solution f_∞ which we can take as a solution to the above differential equation since

$$\frac{\partial f_\infty}{\partial t} = 0 \quad \implies \quad \mathcal{L}f_\infty - \rho = 0. \tag{90}$$

2.4.1 Jacobi method

(To be completed)(slower but parallelisable)
Discretise equation to become $A\vec{f} = \vec{\rho}$. Decompose A into diagonal and remainder: $A = D + R$. Then iterate $\vec{f}^{n+1} = D^{-1}(\vec{\rho} - R\vec{f}^{n})$, using the element-based formula:

$$f_i^{n+1} = \frac{1}{a_{ii}}\left(\rho_i - \sum_{j\neq i} a_{ij} f_j^n\right) \tag{91}$$

2.4.2 Gauss-Seidel method

(To be completed)(faster but not parallelisable)
Discretise equation to become $A\vec{f} = \vec{\rho}$. Decompose A into lower triangular and strictly upper triangular components: $A = L + U$. Then iterate $L\vec{f}^{n+1} = \vec{\rho} - U\vec{f}^{n}$, using forward substitution:

$$f_i^{n+1} = \frac{1}{a_{ii}}\left(\rho_i - \sum_{j=1}^{i-1} a_{ij} f_j^{n+1} - \sum_{j=i+1}^{N} a_{ij} f_j^n\right) \tag{92}$$

2.4.3 Convergence

Why exactly do this work? Once we discretise the system, we're really solving the matrix equation $A\vec{\phi} = \vec{\rho}$, whose solution is $\vec{\phi} = A^{-1}\vec{\rho}$. The discretised initial value problem can be written as

$$\vec{\phi}^{n+1} - \vec{\phi}^{n} = dt(A\vec{\phi}^{n} - \vec{\rho}) \tag{93}$$

$$\vec{\phi}^{n+1} = (1 + dtA)\vec{\phi}^{n} - dt\vec{\rho} \tag{94}$$

$$= B\vec{\phi}^{n} - \vec{r}. \tag{95}$$

where $B = 1 + dtA$ and $\vec{r} = dt\vec{\rho}$. Iterating this gives:

$$\vec{\phi}^{1} = B\vec{\phi}^{0} - \vec{r}, \tag{96}$$

$$\vec{\phi}^{2} = B\vec{\phi}^{1} - \vec{r} = B^2\vec{\phi}^{0} - (B + 1)\vec{r}, \tag{97}$$

$$\vec{\phi}^{3} = B^3\vec{\phi}^{0} - (B^2 + B + 1)\vec{r}, \tag{98}$$

$$\vdots$$

$$\vec{\phi}^{n} = B^n\vec{\phi}^{0} - \left(\sum_{m=0}^{n-1} B^m\right)\vec{r}, \tag{99}$$

In the limit $n \to \infty$, if the absolute value of all the eigenvalues of B are less than 1 (which is ensured if the spacing on the space-time grid is small enough) then $B^n\vec{\phi}^{0} \to 0$ and

$$\vec{\phi}^{n} \to \vec{\phi}^{\infty} = -\left(\sum_{m=0}^{\infty} B^m\right)\vec{r}. \tag{100}$$

Recall the following matrix formula:

$$(1 - T)^{-1} = 1 + T + T^2 + T^3 \cdots . \tag{101}$$

So we have

$$\vec{\phi}^{\infty} = -(1 - B)^{-1}\vec{r} = -(1 - 1 - dtA)^{-1}(dt\vec{\rho}) = A^{-1}\vec{\rho}. \tag{102}$$

Notice $A^{-1}\vec{\rho} = (1 + B + \cdots + B^{n-1})\vec{r} + O(\rho_s^n)$, where ρ_s is the largest eigenvalue of B. Assuming that each iteration reduces the difference between the estimate and the true solution by a factor ρ_s, the number of iterations n required to reduce this difference by a factor 10^{-p} is given by

$$\rho_s^n = 10^{-p}, \tag{103}$$

$$\implies n\ln(\rho_s) = -p\ln(10), \tag{104}$$

$$\implies n = \frac{-p\ln(10)}{\ln(\rho_s)}. \tag{105}$$

For the Jacobi method we have $\rho_s = \rho_J = \cos(\frac{\pi}{N})$. So the number of iterations of the Jacobi method needed to reduce the difference by a factor of 10^{-p} is

$$n_J = \frac{-p\ln(10)}{\ln(\rho_J)} = \frac{-p\ln(10)}{\ln(\cos(\frac{\pi}{N}))}. \tag{106}$$

For the Gauss–Seidel method we have $\rho_s = \rho_{GS} = \cos^2(\frac{\pi}{N})$. So the number of iterations of the Gauss–Seidel method needed to reduce the difference by a factor of 10^{-p} is

$$n_{GS} = \frac{-p\ln(10)}{\ln(\rho_{GS})} = \frac{-p\ln(10)}{\ln(\cos^2(\frac{\pi}{N}))} = \frac{-p\ln(10)}{2\ln(\cos(\frac{\pi}{N}))} = \frac{n_J}{2}. \tag{107}$$

Hence the Gauss–Seidel converges twice as fast as the Jacobi method.

2.4.4 Successive over-relaxation

(To be completed)
Discretise equation to become $A\vec{f} = \vec{\rho}$. Decompose A into lower triangular, diagonal and upper triangular components: $A = L + D + U$. Then choose $\omega \in [1, 2]$ and iterate $(D + \omega L)\vec{f}^{n+1} = \omega\vec{\rho} - (\omega U + (1 - \omega)D)\vec{f}^{n}$, using forward substitution:

$$f_i^{n+1} = (1 - \omega)f_i^n + \frac{\omega}{a_{ii}}\left(\rho_i - \sum_{j=1}^{i-1} a_{ij}f_j^{n+1} - \sum_{j=i+1}^{N} a_{ij}f_j^n\right). \tag{108}$$

2.4.5 Residual

(To be completed)
How precise is our answer after n steps? If ϕ is the true solution to $\mathcal{L}\phi = \rho$ and f is our approximation to ϕ then the true error is $\phi - f$. Or to turn this into a number we can take $\sum_i |\vec{\phi}_i - \vec{f}_i|$. However, we don't know ϕ (it's what we're trying to compute) so we can't compute this error. Instead we can compute the residual:

$$\mathrm{res}(\vec{f}) = \sum_i |(A\vec{f})_i - \vec{\rho}_i|, \tag{109}$$

where A is discretised $\mathcal{L}$. The closer f is to ϕ, the closer $\mathrm{res}(f)$ is to 0.