PHAS0030: Further Practical Mathematics & Computing
Session 8: Random numbers and stochastic methods

David Bowler

March 4, 2019

In this session, we will explore the use of random numbers in modelling physical processes. We will consider the generation of *pseudo*-random numbers by computers, and how to form and use different probability distributions. We will examine the Monte Carlo approach to integration, and see how this can be used to study problems in thermodynamics such as phase transitions, and be adapted to optimise a function.

Contents

1 Objectives

The objectives of this session are to:

- Understand how pseudo-random number generators work
- Explore simple applications of random numbers to stochastic processes
- Examine how different random numbers can be generated from specific probability distributions
- Introduce the Monte Carlo approach to integration
- See how to apply Monte Carlo modelling to the calculation of thermodynamic quantities, and minimisation

2 Review of Session 7

In the seventh session we examined the use of modelling with particles in computational physics. We introduced the Verlet algorithms for energy-conserving, time-reversible integration of Newton's laws of motion, and considered simulation cells as finite domains for modelling. We used the Lennard-Jones potential as an example of a pair potential (one that depends only on the distance between pairs of particles) and looked at initialisation of particle positions and velocities. We then turned to inverse quadratic interactions, exemplified by electrostatics and gravity, and ended with a brief consideration of coarse-graining: applying particle methods to general physical problems.

3 Pseudo-random numbers

There are many areas of physics where events occur in truly random fashion, such as radioactive decay or the Brownian motion of a particle. In order to model such processes, we need to have a way to simulate random events on a computer, which is the same as saying that we need to generate numbers randomly. It turns out that there are certain classes of problem where replacing the original problem with a stochastic one (i.e. a random one) can give us a useful answer. Of course, in these cases we need some statistical measure of the accuracy of the simulation.

There is no way to generate a truly random number on a computer; instead, there are algorithms that allow us to generate a long sequence of *uncorrelated* numbers, which we treat as *pseudo*-random numbers. These sequences are set by a seed number[1] which means that it is possible to reproduce what should be a "random" process: this is important for testing.

[1] This is often taken to be the number of seconds since a particular date, or can use environmental noise from computer hardware.

The simplest pseudo-random number generator is a linear congruential method, where integers in a sequence are generated from the previous entry via multiplication and a modulo operation:

$$I_{n+1} = (A \times I_n + C)\%M \tag{1}$$

where the performance of the generator depends exquisitely sensitively on the integers A, C and M. You can find many suggested combinations of these parameters, but what is important to understand is that a poor choice can lead to correlations between successive numbers (which would mean that *any* simulation based on this would be wrong—potentially disastrously so).

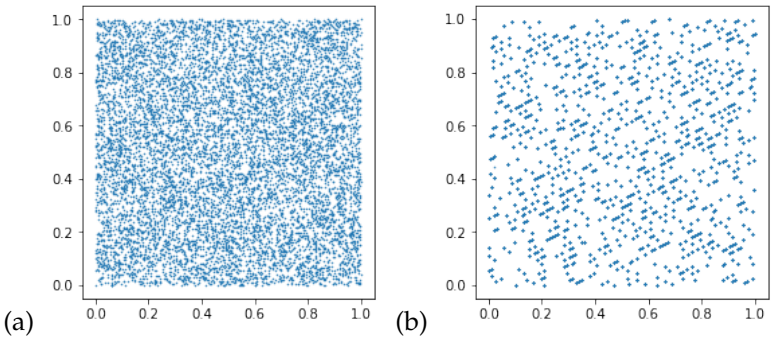

Figure 1: Samples of 10,000 pairs of random numbers plotted to show possible correlations. (a) An uncorrelated sequence. (b) A sequence showing correlations.

A reasonable choice, which shows no correlations under most tests, is $A = 16807$, $C = 0$ and $M = 2^{31} - 1$. If we plot this in two dimensions (using I_j and I_{j+1} as x and y coordinates) then we find something like Fig. 1(a). A different set of numbers ($A = 106$, $C = 1283$ and $M = 60752$) gives strong correlations, as seen in Fig. 1(b)—we should *not* see the diagonal stripes. Note that the maximum value of the integer generated in the sequence is set by the value of M; we have normalised these to lie between zero and one.

One of the most famous problems with random number generators is the RANDU algorithm ($A = 65539$, $C = 0$, $M = 2^{31}$) which shows no obvious correlations in two dimensions, but when plotted in three dimensions (using three successive integers as coordinates) reveals that the numbers lie on planes.

We will use the Numpy `np.random` module, which uses an extremely good pseudo-random number generator (the Mersenne Twister). To generate a single random number from 0 to 1 (with the but not including 1 itself) you should use `np.random.random()`. Other commands that can be used include:

- `(b-a)*np.random.random() + a` to generate numbers from `a` to `b`
- `np.random.rand()` to generate a set of random numbers in a given shape (e.g. a (2×5) array would use `np.random.rand(2,5)`[2])
- `np.random.randint(low,high)` to generate integers between `low` and `high`
- `np.random.seed()` to set the seed, if needed

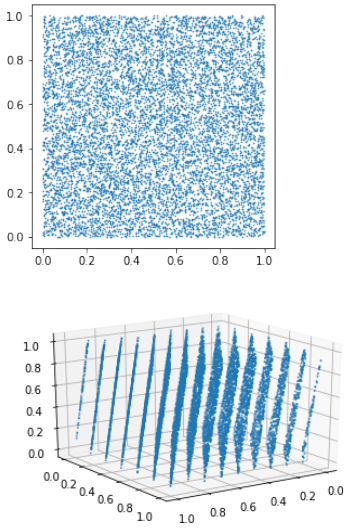

Figure 2: Samples of 10,000 triples of random numbers from the RANDU generator. Top: plotted in $x - y$ plane. Bottom: plotted in 3D.

[2] Note that we do *not* need to pass a tuple here, unlike for `np.zeros()` etc.

There are also many routines to generate random numbers from non-uniform distributions; we will discuss how this might be done below.

If we are modelling an event which has a particular probability of occurring, then in general we will choose a random number between zero and one, and if that number is *less than* the probability, we say that the event has occurred.

3.1 Exercises

In-class

1. One isotope of thallium (^{208}Tl) has a half-life $t_{1/2} = 3.053$ minutes, decaying to ^{208}Pb. Given that the probability of decay in a time interval dt is $p = 1 - e^{-dt/\lambda}$ where $\lambda = t_{1/2}/\log 2$, write a simple `for` loop to track the populations of ^{208}Tl and ^{208}Pb over 1,000 seconds with $dt = 1$s, starting with 1,000 atoms of ^{208}Tl and 0 atoms of ^{208}Pb. At each step, evaluate the possibility that each ^{208}Tl atom decays by creating an array of random numbers from 0 to 1 whose length is the number of ^{208}Tl atoms (use `np.random.rand`). You can then use `np.where` to test whether each of these numbers is less than the probability of decay (`np.where(condition, a, b)` will return `a` if `condition` is true and `b` if `condition` is false; you can pass an array as part of `condition`, e.g. `np.where(inarray>5.0,1,0)` will return an array populated with 1 if the corresponding entry in `inarray` is greater than 5, and 0 otherwise). Sum over the number of atoms that do decay, and use this number to update the total number of thallium and lead atoms. Make a plot of the number of atoms of each species against time.

```
arr = np.arange(10)
print(arr)
[0 1 2 3 4 5 6 7 8 9]
print(np.where(arr>5.0,1,0))
[0 0 0 0 0 0 1 1 1 1]
```

Figure 3: An example of using `np.where`.

2. We will model a random walk on a 2D lattice. Define a side length and create a two-dimensional array to store the number of times each point has been visited (initialise to zero). Define an array of possible neighbouring points using something like `np.array([[-1,0],[1,0],[0,-1],[0,1]])` (be sure that you understand what we are doing, and feel free to use your own implementation). Define a number of steps, setting it to 1,000 to start, and create an array to store the overall trajectory (dimension $2 \times N_{steps}$). Now, starting at the middle of the lattice, loop over the number of steps and at each step pick a neighbouring point at random and move there. Impose hard-wall boundaries ($x < 0 \Rightarrow x = x + 2$ and $x > L \Rightarrow x = x - 2$ and also for y). Increment the value of the visit array by one, and store the next point of the trajectory. You can visualise the random walk using `plt.imshow` for the visits and `plt.plot` for the trajectory. The probability of reaching a certain distance after N steps is given by $P(r) = 2re^{-r^2/N}/N$; you could show this as an image on the same graph as the trajectory (using `np.meshgrid` for the x and y variables).

Further work

1. Make a new piece of code to explore the decay chain $^{218}\mathrm{Po} \rightarrow {}^{214}\mathrm{Pb} \rightarrow {}^{214}\mathrm{Bi} \rightarrow {}^{210}\mathrm{Pb}$ with half lives of 3.1 minutes, 26.8 minutes, 19.9 minutes for the first three nuclei (the final is stable). Use the same approach as before, but you will need to be careful about the order in which you calculate decays: start with the *final* decay and work back along the chain (remember that `range(3,-1,-1)` will produce 3, 2, 1, 0). You will need at least 6,000 seconds. Plot the populations, and ensure that the total number of atoms is conserved.

2. Extend the random walk to three dimensions (you will need six neighbours). I found that a 3D projection using `ax.plot` worked well to visualise the trajectory.

4 *Probability Distributions*

The pseudo-random numbers that we generated above are uniformly distributed between zero and one. But we often want to use numbers that are distributed according to a different probability distribution; the most common is probably the normal distribution.

There are a number of approaches to generate distributions; if you have need of a particular distribution, it is often easiest to use an available function (see Numpy documentation for `np.random` for a list of distributions). However, we will examine one useful result, and two approaches to finding required distributions.

The central limit theorem is a fundamental theorem of statistics, which roughly states that the average of a number of independent random numbers (with well-defined mean and variance) has a normal distribution. In other words, if we sum a sample of random numbers from `np.random.random` and average, and repeat this many times, the resulting set of samples will follow a normal distribution[3]. You will test this in the exercises.

[3] You can get a qualitative feel for why this might happen by thinking about rolling dice: a single die should be completely uniform, generating numbers from 1 to 6 with equal probability; two dice, however, generate numbers with a peak at the middle of the distribution. A fuller proof or justification is beyond the scope of the course.

The *transformation method* relies on some mathematics that the user must do beforehand. If we have a variable x drawn from a probability distribution $p(x)$ and another variable that we can write as $y = f(x)$ then we might ask what the probability distribution of y, $q(y)$, will be. Since the two variables are related, we can say that we must have $p(x)dx = q(y)dy$, so we could say that $q(y) = p(x)/f'(x)$ where $f'(x) = dy/dx$.

But we can go further than this; starting from $p(x)dx = q(y)dy$ and integrating, we can see that:

$$\int_{-\infty}^{x} p(x')dx' = \int_{-\infty}^{y(x)} q(y')dy' \tag{2}$$

Now if x is drawn from a uniform distribution from 0 to 1 (as is common and easy), then the left hand side will evaluate to x. So if we can integrate $q(y)$, then we can find an equation for x in terms

of y; if we can invert this, then we have a formula that will convert a uniform random number (x) into a random number drawn from a specified distribution, $q(y)$.

For instance, if we want to model a Poisson distribution with $q(y) = e^{-y/\lambda}/\lambda$, say, then we can show that $y = -\lambda \log(1-x)$ is the transformation that takes the uniformly distributed x and makes it obey a Poisson distribution. The transformation method relies on being able to integrate and invert the functions.

The *rejection method* allows us to generate a target distribution $q(x)$ from another distribution $p(x)$ so long as $p(x) > q(x)$ for all values of x in the interval we require. We must be able to generate samples from the distribution $p(x)$ (so we often use a uniform or a normal distribution for this)[4]. The algorithm is as follows:

[4] Another condition is that $p(x)$ must be normalisable, though not necessarily normalised.

1. Choose x_i from the distribution $p(x)$
2. Evaluate $z_i = p(x_i)$. Generate a random number, n_i, from $0 \to z_i$
3. If $q(x_i) < n_i$ then accept x_i, otherwise repeat

If we draw a large number of samples x_i using this approach, then they will obey the target distribution, $q(x)$. The efficiency of the method will depend strongly on how much of the area under $p(x)$ is taken by $q(x)$: if this is a large fraction, the method will be efficient, while if it is a small fraction it will be inefficient. A nice illustration of why this method works is to imagine a piece of paper cut into the shape of $p(x)$ with $q(x)$ drawn on it; now make dots at random on the piece of paper, keeping count of the points that lie underneath the curve of $q(x)$. The x-coordinates of this set will be drawn from the target distribution.

4.1 Exercises

In-class

1. Write a function that takes three parameters as input: `Nsamp`, the number of samples; `mu`, the mean; and `sigma`, the width. The function should return the *mean* of N_{samp} uniformly selected random numbers in the range $\mu - \sqrt{3N_{samp}}\sigma \to \mu + \sqrt{3N_{samp}}\sigma$.[5]
2. Now make N_{dist} calls to your function (vary N_{dist} from 10^3 to 10^6, say) using $\mu = 0.0$ and $\sigma = 1.0$ and store them in a table. Plot a histogram using `np.hist` and set the parameters `bins` appropriately and `density=True`. Plot a Gaussian on the same graph to check your function ($e^{-x^2/2\sigma^2}/\sqrt{2\pi\sigma^2}$). Experiment with N_{samp} between 2 and 20 (say).
3. Write a function to implement the rejection method where $p(x)$ is uniform, and $q(x) = (1+x^2)e^{-x^2/2}/\sqrt{8\pi}$. Ensure that $p(x)$ is scaled to be larger than the maximum value of $q(x)$ (around 0.25) and draw samples from $-5 \leq x < 5$. You will need something like a while loop to implement steps 1-3 of the method above. Return an array of samples set by an input parameter.

[5] The factor of $\sqrt{3}$ comes from the variance of the uniform distribution

You should write out how many total tests are made, along with the number of samples.

4. Now test this for 10^5 samples, and plot a histogram of the distribution, along with a line of the target. How accurate and efficient is the method?

Further work

1. Using the transformation method, we can show that a Poisson distribution such as radioactive decay can be generated from a uniform random distribution x by using $y = -\lambda \log(1 - x)$. Use this to generate the lifetimes of 1,000 Tl atoms. Now sort these lifetimes (`np.sort` will return a sorted array) and write a for loop over time from $1 \rightarrow 1000$s, at each step counting the number of Tl atoms that have decayed (i.e. how much further along the sorted array you need to go to exceed the present time). Store the number of Tl and Pb atoms, as before, and plot against time. Notice how we generate the complete decay history in about half the time.
2. Write a function to implement the rejection method for the same function in the in-class exercises, but using $p(x) = Ae^{-x^2/2\sigma^2}/\sqrt{2\pi\sigma^2}$. Set $A = 1.5$ to ensure that $q(x)$ is completely enclosed.
3. Test again, and compare the accuracy and efficiency to the case using the uniform distribution.

5 *Monte Carlo integration*

Monte Carlo methods[6] use points selected from a probability distribution function to perform integrals of one kind or another. A simple way to understand this is to imagine a shape that cannot be integrated whose area you want to know; place a square (say) around it and select points at random from within the square. The fraction of points that lie within the shape give the area.

[6] These are indeed named after the famous part of Monaco known for its casino.

More formally and generally, we choose points at random from within a multi-dimensional integration domain, sum over the value of the integrand and then scale by the average volume[7] per sampling point. We can write:

[7] In however many dimensions are appropriate

$$I = \int \mathrm{d}\mathbf{r} f(\mathbf{r}) \simeq \frac{V}{N} \sum_{i=1}^{N} f(\mathbf{r}_i) \tag{3}$$

with $\mathbf{r}_i$ chosen *uniformly* from within the domain V. It can be shown that the error in the integral scales as $1/\sqrt{N}$ *regardless* of the dimension of the problem.

Why would we want to take this approach, rather than use one of the highly-optimised numerical approaches we saw in Session 3? The answer lies in the dimensionality that we mentioned above: numerical integration methods scale poorly with dimension, while Monte Carlo methods scale extremely well. One area where they

have found many applications is statistical mechanics, where the calculation of a partition function requires the integral over all coordinates of N particles: a $3N$-dimensional integral. With only five points in each dimension and ten particles, this would be 5^{30} points, which is not a practical calculation.

However, uniform sampling is often a very bad choice. If we consider the partition function then we are sampling the function $\exp(-E/k_BT)$, where $E = E(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$. The important part of this function[8] forms only a small part of its space, so uniform sampling will waste a lot of time and computational resources. We turn to a method called *importance sampling* which shares similarities with the methods we used above to generate non-uniform distributions. We write:

[8] i.e. where $E \sim k_BT$, before the exponential becomes essentially zero

$$\int_a^b f(x)dx = \int_a^b \frac{f(x)}{p(x)} p(x)dx \tag{4}$$

and note that $p(x)$ is a normalised probability distribution. Then if we discretise, we find:

$$I \simeq \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} \tag{5}$$

where the points x_i are drawn from the normalised distribution $p(x)$. The trick, of course, is to find a good distribution to use that samples more heavily in the areas where the function has a large value and less heavily in the areas where it has small values. This can significantly improve the convergence of the integral with number of samples.

5.1 Exercises

In-class

1. Write a (very simple) function to calculate the integrand $x^{-1/3} + x/10$
2. Use Monte Carlo integration to evaluate the integral between 0 and 1 using different numbers of samples (between 10^3 and 10^6) and compare to the analytic result: 1.55.
3. Now use importance sampling: if y is drawn from a uniform distribution from 0 to 1, then writing $x = y^{3/2}$ and using $p(x) = 2x^{-1/3}/3$ will give an improved estimate of the integral for a given number of samples. Alter your code above to use this, and investigate the number of samples needed to converge.

Further work

1. The equation of an ellipse is $x^2/a^2 + y^2/b^2 = 1$, and the corresponding area is $A = \pi ab$. Use the Monte Carlo integration technique to evaluate the area of an ellipse by drawing samples uniformly from (a) a square that encloses the ellipse and (b) a rectangle that encloses the ellipse.

6 Monte Carlo simulation

A standard result in thermal physics or statistical mechanics is that for a set of distinguishable particles at temperature T, the probability that a particle will occupy the energy level E_i is given by:

$$p(E_i) = \frac{e^{-E_i/k_BT}}{\sum_i e^{-E_i/k_BT}} \tag{6}$$

As we saw with the work on molecular dynamics in Session 7, it turns out that the properties of a system can be modelled with relatively few particles. With the molecular dynamics method, we found time averages of quantities by integrating[9] Newton's equations of motion. With the Monte Carlo method, we can take *ensemble* averages of quantities by sampling from distributions.

[9] We also say that we *evolve* the system forward in time; the two terms describe the same process, as we integrate to move forward in time.

What we will model here is the approach to equilibrium (which is exactly what we did with molecular dynamics: after creating an initial set of positions and velocities, we then needed to equilibrate), by randomly exchanging energy between particles. This method dates back over fifty years, and is often known as the Metropolis method[10].

[10] Metropolis et al., J. Chem. Phys. **21**, 1087 (1953)

The basic idea is to start with a random distribution of particles over energy levels (which are often determined by particle positions, or spins, or other physical properties). We then select a particle at random and alter its energy (again, generally at random, unless there are only two energy levels). If the energy goes down, then we accept the move; the key point of the Metropolis algorithm is that there is a finite chance to accept a move if the energy goes up. We use the probability $p = e^{-\Delta E/k_BT}$ where the energy change is ΔE, so that the probability decreases rapidly as the energy change increases. This process means that the system can climb out of local minima (but not the global minimum), with the probability of escaping from a minimum increasing as the temperature increases (just as is seen in nature).

We will use this relatively simple procedure to model the behaviour of the Ising model: a model system for interacting spins on a lattice which is widely studied. We assume a square lattice with spin-1/2 particles that interact only with their nearest neighbours, and potentially an external magnetic field. The energy in this case is given by:

$$E = -\sum_{i,j} s_i s_j J - \sum_i Bms_i \tag{7}$$

where the spins s_i take on values of 1 or -1, $m = 1/2$ and J is the quantum mechanical exchange coupling between spins. We will characterise the system in terms of the ratios J/k_BT and mB/k_BT.

To set up a simulation, we start with a random array of spins on a lattice (which should then have a net spin of close to zero). We then iterate the following algorithm:

1. Choose a particle at random

2. Calculate the change in energy when its spin is flipped (note that this is only for this spin, and only requires its neighbours) using Eq. (7)

3. If the energy goes down, accept the move

4. If the energy goes up, accept the move with probability $p = e^{-\Delta E/k_B T}$ (in other words, choose a random number from 0 to 1 and accept the move if this number is *less than* p)

The system will have periodic boundary conditions (easily implemented with modulo, as we saw last week), and we will need to monitor the total energy and the average spin (this is a measure of long-range order, which is important to monitor in this kind of simulation). We will test two cases: first, equilibration given an external field but zero coupling between spins (i.e. $J = 0$), equivalent to a paramagnet; second, equilibration with no external field but varying coupling between spins, which will display ferromagnetic behaviour given strong enough coupling.

6.1 *Simulated annealing*

We can apply the Monte Carlo method to the minimisation of functions: if we start at a high temperature, and equilibrate, and then gradually reduce the temperature, equilibrating at each stage, it is quite likely[11] that we will find the minimum of the function. This is a direct analogy of the process used to remove defects from metals and glasses[12]. The process is simple to implement, but finding the details of how to reduce the temperature, and how long each run at a fixed temperature should be, is extremely difficult. We will examine a simple example in the further work.

[11] Note that, as with all optimisation procedures, we cannot guarantee to find the global minimum.

[12] And, bizarrely, chocolate: the process is known as tempering as well as annealing.

6.2 *Exercises*

In-class

1. Define a box length (set it to 50 for now), and set `B_over_kT = 0.4` and `J_over_kT = 0.0`. Create an initial lattice of spins using the command:

```
(-1)**np.random.randint(0,2,size=(boxlen,boxlen))
```

   Notice that the `np.random.randint` command will return a two dimensional array of numbers which are either 0 or 1; using this as the exponent gives us 1 or -1.[13] Plot your array using `plt.imshow` to ensure that it is correct—you should see something like Fig. 4.

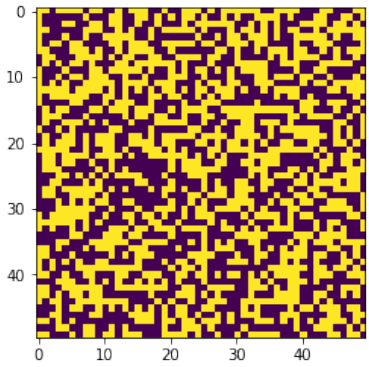

Figure 4: An example set of random spins.

[13] It is very easy, as I did, to omit the brackets around -1 which gives a uniform array with value -1; check that you understand why.

2. Use `np.roll` to create an array which holds the sum over all neighbouring spins for every spin on the lattice. You can use this to evaluate the total energy using:

```
etot = -np.sum(spins*(B_over_kT + J_over_kT*sum_neigh))
```

Check that you understand why this is equivalent to Eq. (7).

3. Write a function to perform a trial update on the spin array. It should take the location of the spin as input parameters, evaluate the energy *change* if the spin flips (you should calculate the positions of the neighbouring spins using modulo) and accept it (change the sign of the spin) according to the criteria given in the algorithm above.

4. Now perform a 50,000 step Monte Carlo simulation. At each step, choose a lattice point at random and call your update function. Keep a record of the overall total energy, average spin and the square of the average spin. Every 5,000 steps (or more often if you want) plot the sample, using subplots (`fig = plt.figure(); ax = fig.add_subplot(3,4,index); ax.imshow()` or similar).

5. Plot the long-range order (average spin) over time and, on a separate plot, the energy. You could also calculate the overall average spin and standard deviation.

Further work

1. Run another Monte Carlo simulation, this time for a box length of 100, setting `B_over_kT` to zero, and experimenting with values of `J_over_kT` between 0.1 and 2. You should find a phase transition when $J/k_BT > 0.46$ with domains of parallel spin emerging in the domain.

2. Implement a simple simulated annealing process to find the minimum of the function $f(x) = 0.001x^2 + \cos(x)e^{0.1(x-x^2)}$. Define a starting temperature $T = 100$, and starting position $x = -5$. Perform a double loop: for each value of temperature, repeat 1,000 times a Monte Carlo trial where you change x by a value drawn randomly from -1 to 1 (accept with the usual criteria); then scale the temperature by 0.9 and repeat. Continue until the temperature is less that 0.01, and output the location and value of the minimum. Test the effect of changing the starting temperature and starting point.

7 *Progress Review*

Once you have finished *all the material* associated with this session (both in-class and extra material), you should be able to:

- Use random numbers to simulate stochastic processes in nature (e.g. radioactive decay)
- Understand how different probability distributions can be created from a uniform distribution

- Use Monte Carlo integration for complex integrals
- Apply Metropolis Monte Carlo to follow systems to equilibrium and to optimise functions.