Random Numbers, Random Walks

Random numbers should not be generated with a method chosen at random.

Donald Knuth, The Art of Computer Programming.

8.1 Introduction

There are many processes in nature that consist of a series of effectively random events. For example, radioactive decay or the division of bacteria cells. There are also processes that can be modelled by selecting values from a known statistical distribution, for example the heating of gas particles in a container. In these last two labs, we introduce an important family of numerical techniques called Monte Carlo simulations that can be used to model situations that involve random events.

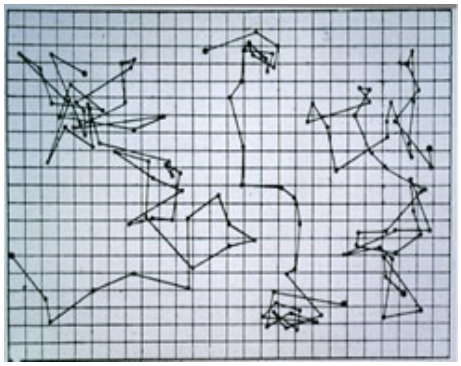


Figure 8.1: The notebook records of Jean Perrin showing the irregular motion of particles suspended in water.

In 1827 the botanist Robert Brown used a microscope to observe particles from pollen suspended in water. He noticed that they jiggled around, seemingly randomly, but did not know what was causing the motion. The effect was named *Brownian motion*.

In 1905 Albert Einstein proposed that the jiggling could be explained by thermal molecular motions. Jean Perrin's work (for which he won the 1926 Nobel Prize) confirmed this experimentally, and provided a way of calculating the size of molecules.

In this lab we will investigate random numbers and use them to generate one dimensional random walks based on the idea of Brownian motion.

8.2 Lab objectives

By the end of this lab sessions you should be able to:

1. Generate random numbers using MATLAB;
2. Generate random numbers from different probability distributions;
3. Run random walk simulations to model a system.

If you still don't know how to do any of these things once you have completed the lab, please ask your tutor for help. You should also use these lists of objectives when you are revising for the end of semester exam.

8.3 Walkthrough: Random Numbers

You can generate random numbers in MATLAB using the **`rand`** function. **`rand`** returns a uniformly distributed pseudo-random number in the range $0 \leq x < 1$. For example:

```
>> rand()
ans = 0.2785
```

You can also get the same result calling **`rand`** without the parentheses. If you run **`rand`** multiple times, you will notice that the number is different each time:

```
>> rand()
ans = 0.9575
>> rand()
ans = 0.9649
>> rand()
ans = 0.1576
```

You can also generate a vector or matrix of random numbers by passing in the vector dimensions:

```
>> rand(2,3)
ans = 0.8491    0.6787    0.7431
      0.9340    0.7577    0.3922
```

Experiment to see what happens if only a single argument is given to **`rand`**?

Result is a square matrix. This can be a problem if you write rand(1e6) instead of rand(1e6,1)!

Now we mentioned that these are actually *pseudo-random* numbers from a *uniform* distribution. What do each of these terms mean?

8.3.1 Pseudo-randomness

Although these numbers seem random, MATLAB is using a sophisticated algorithm to generate each successive 'random' number, and so they are not truely random. Theoretically, MATLAB can generate over 2^{1492} (about 10^{450}) numbers before repeating itself. Each time you start up a MATLAB session, the random number sequence begins at the same place. You can change the starting point of the sequence (a process known as *seeding*), by invoking the statement **`rng(n)`**, where **`n`** is any integer. By default, **`n`** is set to zero when MATLAB starts.

For example, let's seed the random number generator with $n = 5$, and then generate some numbers:

```
>> rng(5)
>> rand()
ans = 0.2220
>> rand()
ans = 0.8707
```

Now let's re-seed the number generator and generate some more numbers:

```
>> rng(5)
>> rand()
ans = 0.2220
>> rand()
ans = 0.8707
```

You can see these numbers are identical to before, demonstrating that our random number generator is not truely random. This is why computationally generated random numbers should really be called *pseudo-random* numbers.

8.3.2 The uniform distribution

The random numbers generated by **`rand`** are drawn from a *uniform* distribution. That means that if you divide the range 0 to 1 into equal bins, there should be an equal number of numbers in each bin. You can check if this is true by generating a set of random numbers and then plotting a histogram of their values:

```
>> numbers = rand(1, 100)
>> hist(numbers)
```

You'll probably find that the plotted distribution is somewhat even, but maybe not as even as you expected. The reason is that numbers drawn from a uniform distribution will be uniformly distributed *if we draw enough of them*. For example, if we generate a 10 000 numbers instead of 100, the distribution should be much more uniform.

```
>> numbers = rand(1, 10000)
>> hist(numbers)
```

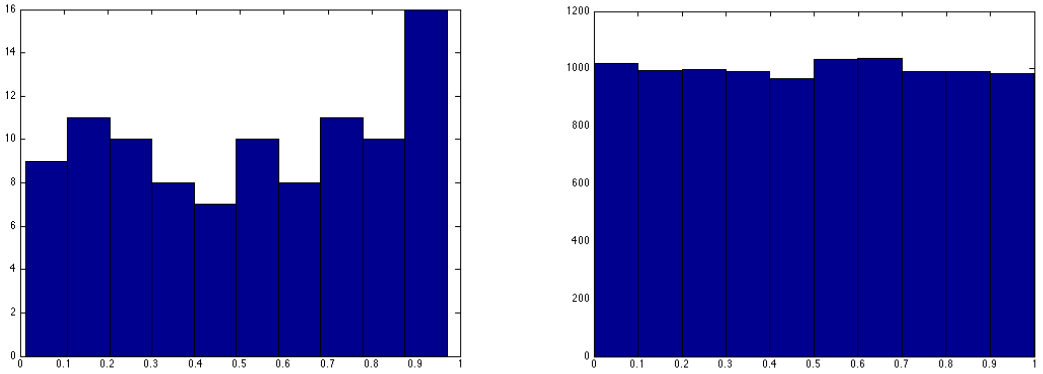


Figure 8.2: A histogram of 100 (left) and 10 000 (right) random numbers drawn from a uniform distribution.

8.4 Walkthrough: Monte Carlo simulations

A Monte Carlo simulation is one in which repeated random sampling is used to obtain a numerical solution to a problem. Usually many iterations of the simulation are run, resulting in a probability distribution for some unknown entity (for example, the probability that a neutron will escape the shielding around a radiation chamber).

Monte Carlo was a secret code name chosen by Nicholas Metropolis, who was one of the scientists who developed Monte Carlo methods as part of the Los Alamos nuclear weapons project in World War II. It was named after the casino, where the uncle of another of the inventors, Stanislav Ulam, used to gamble.

To demonstrate a simple Monte Carlo simulation, we can use the example of a tossing a coin. Say we want to answer the question: *If a toss a coin a large number of times, what fraction of the time will it come up heads?* With such a simple case, we could just answer this question analytically. But imagine we didn't know how to do that. An alternative would be to actually toss a coin thousands times, but this would take a long time. A Monte Carlo simulation lets us do this experiment computationally.

We start by modelling a single coin toss. There are two options when you toss a coin, and both are equally likely. So we can say that if we generate a number x between 0 and 1, then we will call it **heads** if $x < 0.5$ and **tails** if $x \geq 0.5$. One way of implementing this is to use **`round`**, and make heads equal to 0 and tails equal to 1:

```
>> x = round(rand())
x = 0
>> x = round(rand())
x = 1
```

Now we know how to toss a single coin, we can easily simulate tossing a coin 100 times by generating a vector with 100 random numbers in:

```
>> n = 100;
>> x = round(rand(1, n));
```

We could then count the number of heads by working out how many elements of **x** are equal to 0:

```
>> nheads = length(x(x==0))
nheads = 44
```

If we had just done this experiment in real life (physically tossing a coin 100 times) we'd probably be pretty tired. We would conclude that since heads comes up 44 times out of 100, the probability of heads $P(h)$ is:

```
>> pheads = nheads / n
pheads = 0.4400
```

But there is a problem with this conclusion. Say our friend has also been tossing a coin, and after 100 tosses she has found there are 52 heads. Her conclusion is that $P(h) = 0.52$. Another friend claims to have got 49 heads out of 100 coin tosses and concludes that $P(h) = 0.49$. What is the correct answer?

8.4.1 A probability distribution

What we obtain from a simulation like this is not an exact answer, but a probability distribution. We can repeat the coin toss experiment many times using a **for** loop, adding the calculated probability to a vector in each iteration. You should save this code as `coin_toss.m` so you can run it multiple times.

```
p = [];
n = 100;
for iters = 1:1000
  x = round(rand(1, n));
  pheads = length(x(x==0)) / n;
  p = [p pheads];
end
```

Running this code and plotting a histogram of these probabilities:

```
>> coin_toss
>> hist(p, 0:0.05:1)
```

shows a range of values centred on 0.5 (see Figure 8.3). The peak of this distribution (0.5) is the best estimate of the probability of getting heads in a coin toss experiment. Note that you might have to adjust the histogram bin range and plot axis range to get a clear plot.

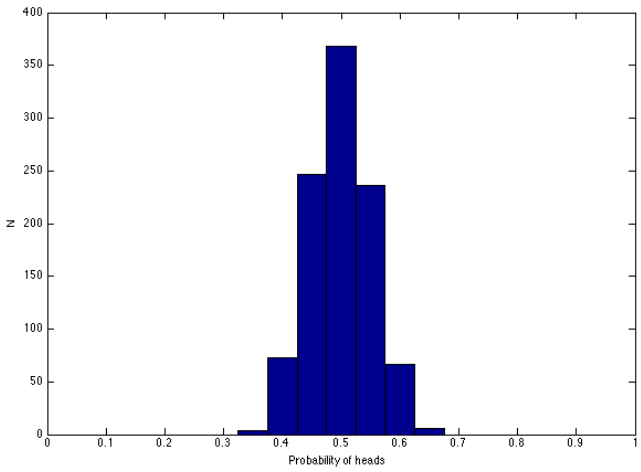


Figure 8.3: Distribution of probabilities after repeating the coin toss experiment 1000 times.

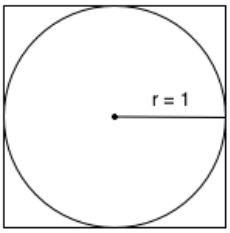
The power of Monte Carlo methods is that we can run simulations that are impractical to do in real life.

8.5 Activity: Calculating π

In this activity we will look at a clever and yet simple way of estimating π using a Monte Carlo simulation.

Question 1
Imagine you have a circle inscribed inside a square, as shown. Derive an expression for π in terms of the area of the square (A_s) and the area of the circle (A_c).

$A_c = \pi r^2$
Implies $\pi = A_c/(r^2)$ **(Eq. 1)**
$A_s = (2r)^2$
Implies $r = \sqrt{(A_s)}/2$ **(Eq. 2)**
Substitute (2) into (1)
$\pi = 4A_c/A_s$



Question 2
Write MATLAB code to generate two vectors `x` and `y` that both contain 100 random numbers with values between -1 and 1. Check the numbers appear random by plotting x vs. y. Write your code below.

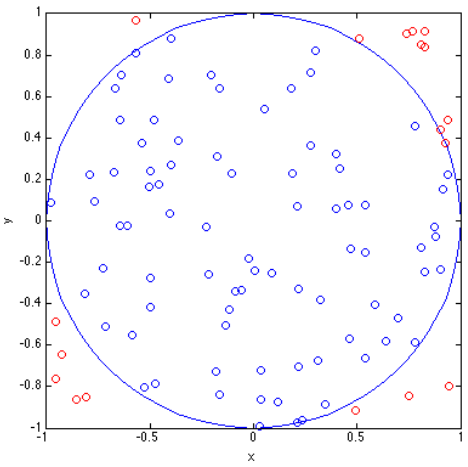
```
n = 100;
a = -1;
b = 1;
x = a + (b-a)*rand(n,1);
y = a + (b-a)*rand(n,1);
```

Tutor note: you need to multiply by 2 so the numbers span the range 0-2, then shift to offset to -1.

Question 3
Now imagine an experiment in which you throw n darts randomly at the square. All of the darts fall inside the square ($n_s = n$), but only some fraction of them fall inside the circle (n_c).

Assume the circle has a radius of 1. Write a program to simulate throwing 100 darts at the square. To do this, generate 100 random numbers between -1 and 1 in each coordinate (x and y). Plot a circle of radius 1, and on the same axes plot the positions of the 100 darts. Sketch your plot below or show it to your tutor.

Hint: It would be useful to plot the darts that fall inside the circle one colour, and the darts that fall outside the circle in a different colour. You can plot a circle using `plot(sin(0:0.1:2*pi+0.1),cos(0:0.1:2*pi+0.1))`.



Question 4
Increase the number of darts to 100000 and calculate the number of darts that fall within your circle, and from this, provide an estimate of π using the result you derived in Question 1:

$$\pi = 4\frac{n_c}{n_s} \tag{8.1}$$

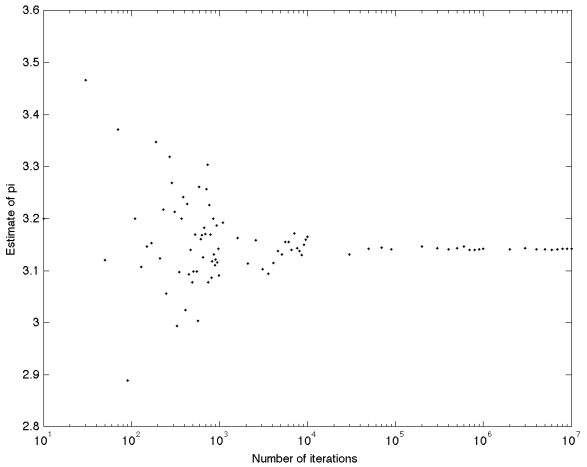
Estimate should be somewhere between 2.9 and 3.3 (of course there is a small chance it is higher or lower if they've only run their code once).
Tutor note: even though PHYS2011 students don't have to do the next question, it would be good to mention, when doing the marking, how they could improve their estimate of pi.

If you want to get a more accurate estimate of π, try increasing the number of darts to, say, 100 000.

Question 5 (PHYS2911 and PHYS2921)
The estimate of π provided by this method is not very accurate. One way of improving this is to increase the number of iterations in our simulation.

Modify your program so that it loops through a range of values for the number of darts, n. For example, try using code like `n =10.^[2:0.2:7]`. Use `pause(0.1)` in your loop so you can see each iteration.

Plot the estimate of π versus the number of iterations, and sketch it below (you probably want make the x-axis on a log scale, using `semilogx`). Include a horizontal line showing the true value of π.



Checkpoint 1:

8.6 Walkthrough: The Normal Distribution

In this section we will use Monte Carlo simulations to model a *random walk* which can be used to represent Brownian motion and particle diffusion.

The normal (or Gaussian) distribution is one of the most important probability distributions and it appears in many different contexts. You have probably already seen this distribution in first year statistics. The probability density function for a normal distribution is

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(x-\bar{x})^2}{2\sigma^2}} \tag{8.2}$$

where $\bar{x}$ is the mean, and σ is the standard deviation of the distribution.

We have previously obtained random numbers from a *uniform* distribution using the **`rand()`** function in Matlab. To obtain random numbers from a Gaussian distribution with $\bar{x} = 0$ and $\sigma = 1$, you can use the **`randn()`** function:

```
>> x = randn(1,10000);
>> hist(x, 100);
>> xlim([-40,40])
```

The left panel in Figure 8.4 shows a histogram of 10000 numbers generated using **`randn()`**. In the above code, the command **`xlim([-40,40])`** sets the limits of the x-axis.

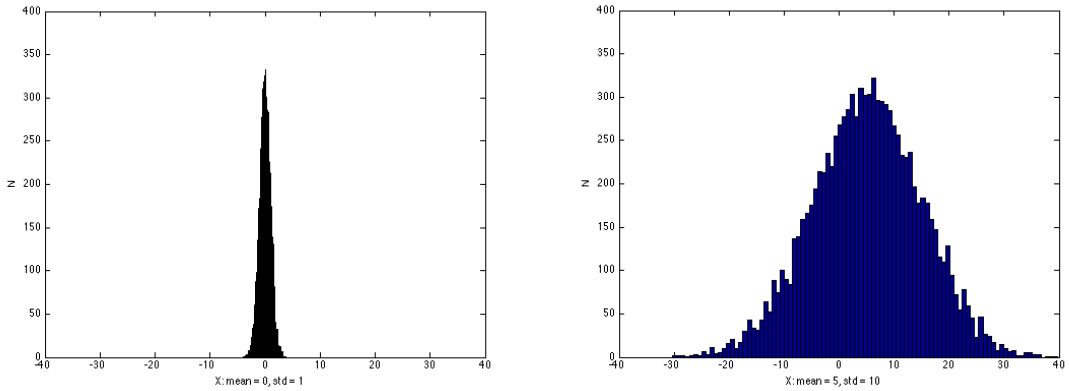


Figure 8.4: Two normal distributions. Left: $\bar{x} = 0$ and $\sigma = 1$. Right: $\bar{x} = 5$ and $\sigma = 10$. For comparison, both are plotted with the same axis ranges.

We can scale the output from **`rand()`** in order to obtain numbers from a normal distribution with, say, a mean $\bar{x} = 5$ and a standard deviation $\sigma = 10$:

```
>> xbar = 5;
>> sigma = 10;
>> x = xbar + sigma.*randn(1,10000);
>> hist(x, 100);
>>xlim([-40,40])
```

See the right panel in in Figure 8.4. Notice that the peak of the distribution is now at **`x = 5`**, and the distribution is wider than before because the standard deviation is higher.

We can calculate the mean and standard deviation of the resulting distribution using:

```
>> mean(x)
ans =    5.0540
>> std(x)
ans =   10.1361
```

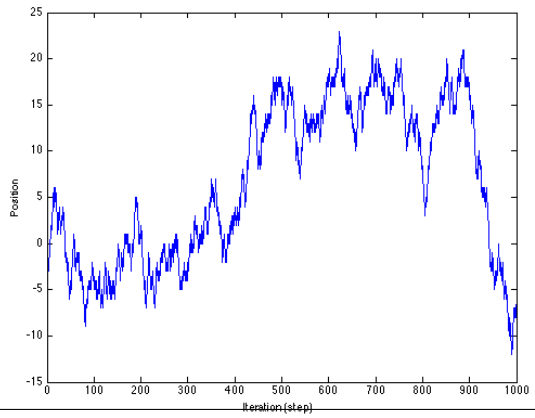
You should see that these results agree (approximately) with the values of $\bar{x}$ and σ you used to create the distribution.

8.7 Activity: Random Walks

A random walk is a path that is formed by taking a series of random steps. The next position in the path depends only on the current position, which makes a random walk an example of a *Markov process*. Brownian motion, the random motion of particles suspended in a fluid resulting from their collision with the fluid molecules, can be modeled using a random walk.

Question 6
You can generate a 1D random walk by considering a particle (e.g. a grain of pollen) that starts out at position $y = 0$ and at each time step (on the x-axis) takes a step of $+1$ or -1 units with equal probability.

Write a program to plot the path taken by the particle over 1000 steps. In each step you should randomly choose whether the particle moves up by one unit (+1) or down by one unit (−1). Sketch your plot below.



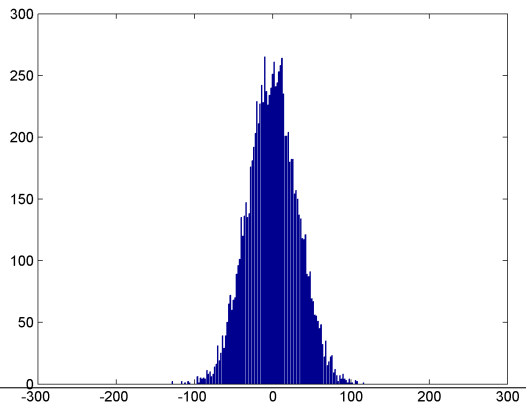
Question 7
Now convert your program to a function **`walker_1d`** that takes a number of steps **`nsteps`** as an input argument, and returns the position of the particle after taking that many steps. You can run your function like this:

```
>> position = walker_1d(1000)
position =   -26
```

(Note your answer will probably be different each time, since the walk is *random*.)

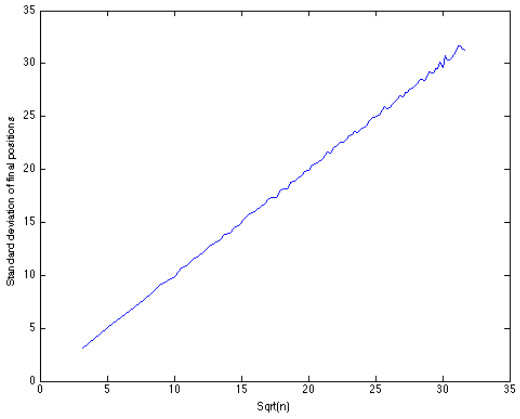
Question 8
Write a program that calls your **`walker_1d`** function to simulate 10 000 particles each taking 1000 steps. Plot a histogram of the final position of the particles (use **`hist(x,-200:200)`**) and sketch it below. What is the expected or mean final position of the particles?

Peak is at zero (mean).



Question 9 (PHYS2911 and PHYS2921)
The characteristic distance that the particle has moved (with respect to its starting point) after n steps is given by the standard deviation. Calculate the standard deviations of the final positions for 10 000 particles for a different number of steps `n = 10:10:1000`

Plot σ vs $\sqrt{n}$ and sketch your plot below.



Question 10 (PHYS2911 and PHYS2921)
What is the relationship between the characteristic distance travelled and n? How many more steps are needed for a particle to travel twice as far as it does with n steps.

$\sigma \propto \sqrt{n}$
Twice as far requires $4n$ **steps.**

Question 11 (PHYS2911 and PHYS2921)
The relationship you found in Question 10 was not known when Brownian motion was first observed, which was a source of confusion for scientists. At the time, scientists tried to measure the speed of diffusion (what might be thought of as the 'overall' speed of the particle) by dividing the change in position by time.

By thinking carefully about your answer to the previous question, what was the apparent paradox that they observed?

Speed of an individual particle does not change in elastic collisions; however the characteristic distance a particle has moved in n steps is σ, so its apparent speed is proportional to proportional to

$$\sigma/n \propto \sqrt{n}/n = 1/\sqrt{n}$$

, i.e., its apparent speed decreases as time goes by.

Checkpoint 2:

8.8 Extra information and references

A copy of Brown's original 1827 manuscript describing what was later called Brownian motion is available here:
`http://sciweb.nybg.org/science2/pdfs/dws/Brownian.pdf`

Einstein's 1905 paper (translated into English) is available here:
`http://users.physik.fu-berlin.de/~kleinert/files/eins_brownian.pdf`