Random processes and Monte Carlo methods

Some process in physics are random (e.g. radioactive decay), others can be treated as random although they are not truly random (e.g. Brownian motion). Here we will look at computational methods for studying random physical processes, both the truly random and the apparently random. To mimic randomness, we need the computer to have an element of randomness and for that we need random numbers.

Random numbers

Technically, computers produce **pseudorandom numbers**. They are actually produced by a deterministic formula referred to (inaccurately) as a **random number generator**. Consider the following equation:

$$x' = (ax + c) \text{ mod } m$$

where a, c and m are integer constants and x is an integer variable. Given a value for x, this equation takes that value and turns it into a new integer value x'. Now we suppose we take that new value and plug it back in on the right-hand side of the equation again and get another value, and so on, generating a stream of integers. Here is a program to calculate the first 100 numbers in the sequence with certain values for the constants. This program plots the numbers as dots on a graph:

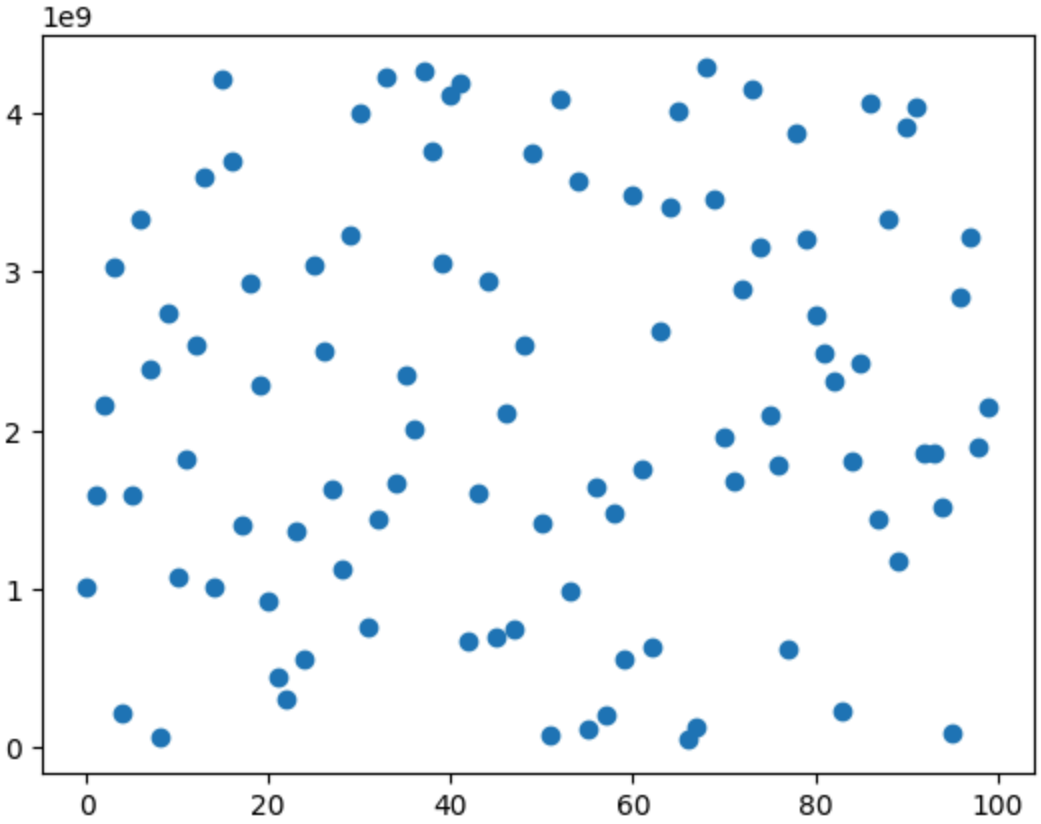
In [2]:
```
from matplotlib.pyplot import plot,show

N = 100
a = 1664525
c = 1013904223
m = 4294967296
x = 1

results=[]

for i in range(N):
    x = (a*x+c) % m
    results.append(x)

plot(results, 'o')
show()
```



As you can see, the positions of the dots look pretty random. This is a **linear congruential random number generator**. It is probably the most famous of random number generators. Please notice the following:

1. **The program is completely deterministic**, so the numbers are not really random. If we run the program twice, it will produce exactly the same graph. Nevertheless, the numbers are random enough for physics calculations.
2. The numbers generated are always positive (or zero), and always less than m. Thus, **all the number generated fall into the range from 0 to m-1**. Often we would like to have numbers in some other range, in which case we must scale them accordingly. For example, if we would like to generate random numbers r in the range $0 \leq r < 1$, we divide the number x obtained above by m.
3. The **values for the constants a,c and m matter**. They look as if they were arbitrarily chosen, but they were chosen with some care. These particular values have been widely tested and are known to work well. It is clear that some other choices are not good. For example, if c and m were both even, then the process will generate only even number or only odd numbers, but not both which is obviously not very random.
4. For a particular choice of constants a, c and m, you can still get different sequences of random numbers by choosing different starting values for x. The initial value is called the **seed for the random number generator**, it specifies where the sequence will start.

The problem with this linear congruential random number generator is that **there are some correlations between the values of successive numbers**. These correlations can **introduce errors into physics calculations that are of significant size**, yet hard to detect. For serious work in computational physics it is important that we use high quality random numbers.

There are many options developed over the years, but one of the generators of choice for physics calculations seems to be nowadays the **Mersenne twister**, which is a **generalized feedback shift-register**

generator. It is difficult to program, but **Python's random package** provides us with a version. These package contains very useful functions such as:

- random(): gives a random **floating-point number uniformly distributed** in the range from zero to one, including zero but not one ($0 \leq r < 1$).
- randrange(n): gives a random **integer** from 0 to n-1.
- randrange(m,n): gives a random **integer** from m to n-1.
- randrange(m,n,k): gives a random **integer** in the range m to n-1 in steps of k.

All these functions generate their numbers using the Mersenne twister and they are consider good enough for serious physics calculations. Everytime you call them they will give you a different answer. For example:

In [6]:
```
from random import randrange

print(randrange(10))
print(randrange(10))
print(randrange(10))
```

```
4
1
7
```

If you would like to use the same random number twice in your program, then you will have to store it in a variable so you can come back to it:

In [7]:
```
from random import randrange

z = randrange(10)
print('The random number is ', z)
print('That number again is ', z)
```

```
The random number is  8
That number again is  8
```

Example:

1. Write a program that generates and prints out two random numbers between 1 and 6, to simulate the rolling of two dice.

2. Modify your program to simulate the rolling of a dice a million times and count the number of times you get a double six. Divide by a million to get the fraction of times you get a double six. You should get something close to 1/36.

Random number seeds

As we saw in the case of the linear congruential generator, a random number generator can have a seed (input value that tells the generator where to start its sequence). The seed specifies not only where the sequence starts but the entire sequence, since once the first number is fixed the rest of the sequence follows automatically. The seed in the case of the linear congruential generator is equal to the first number in the sequence, but this is not always the case. **For many generators, the seed specifies the starting point**

indirectly and it is not actually equal to its first value, yet the seed fixes the entire sequence. The Python version of the Mersenne twister is seeded with the function *seed* from the package *random*, which takes an integer seed as argument. Example:

In [9]:
```
from random import randrange, seed

seed(42)
for i in range(4):
    print(randrange(10))
```

```
1
0
4
3
```

If you run the program again, you will get the exact same sequence.

Why would you want to do this? It is useful for getting your program working in the first place. When you are working with random programs one of the most frustrating things is that you run the program once and something goes wrong, but then you run it again, without changing anything, and everything works fine. This happens because the program does not do the exact same thing every time you run it. For example, there might be an issue with the particular sequence of random numbers generated in the first time around that caused your program to malfunction. This kind of behaviour can make it hard to track down problems. Seeding the random number generator gets sround this problem. Once you get everything working properly, you can get rid of the seed.

Probabilities and biased coins

It happens frequently when writing computer programs for physics calculations that you want some event to take place randomly with probability p. This kind of behaviour is straight forward to create. You generate a random number between zero and one, and you make the event happen if the resulting random number is less than p. Thus, to make something happen with probability 0.2, we might write:

In [10]:
```
from random import random

if random() < 0.2:
    print("Heads")
else:
    print("Tails")
```

```
Tails
```

Since the function random produces a random number uniformely distributed between zero and one, its chance of being less than 0.2 is by definition 0.2. So this program will print "Heads" one fifth of the time (at random), and "Tails" the rest of the time. This coin is obviously biased.

Example: decay of an isotope

The radioisotope ^{208}Tl (thallium 208) decays to stable ^{208}Pb (lead 208) with a half-life of 3.053 minutes. Suppose we start with a sample of 1000 thallium atoms. Simulate the decay of these atoms over time, mimicking the randomness of that decay using random numbers.

Remember that: the number of atoms fall exponentially over time according to the standard equation of radioactive decay:

$$N(t) = N(0)2^{-t/\tau}$$

where tau is the half-life.

In [ ]: