Lecture VII : Random systems and random walk

I. RANDOM PROCESSES

In nature, no processes are truly deterministic. However, while dealing with many physical processes such as calculating trajectories of large objects such as planets, cannonballs or while calculating the current through a simple circuit, we can to a good approximation ignore all the random effects brought in by random processes such as turbulence and electrical heat loss.

In certain systems, however, we cannot ignore these random effects as they are what determines the outcome of the process. Some examples to such random processes are :

- motion of molecules in a solution
- mixing of gases
- electron diffusion in metals
- stock market fluctuations
- cluster growth (snowflakes, microclusters)

Suppose we put a drop of milk in a cup of coffee. If we were to zoom in and follow the motion of a single milk molecule as the molk spreads and diffuses inside the coffee, we would see that it moves in short straight trajectories until it encounters other coffee or milk molecules. After each encounter, we would see that it would change direction in a drastic and unpredictable manner and continue on until the next encounter. Even if we knew all the laws governing the motion of each of the milk and coffee molecules, it would be practically impossible to write down all of the billions of equations of motion and solve them simultaneously. Instead, we simplify the description of the motion of molecules as a *probabilistic* model, which entails the assignment of probabilities to motion of the motion of the molecules in each dimension. In simulating such a process, we would of course need a program to generate the random numbers that we need.

II. WORKING WITH RANDOM NUMBERS IN `OCTAVE`

In `Octave`, there are two separate random number generators.

- `rand` that generates *uniformly distributed* random numbers between 0 and 1.
- and `randn` that generates *normally distributed* random numbers.

Numbers chosen from a *uniform* distribution all have equal probabilities of occuring. The simplest process with a uniform distribution is a coin toss where the probability of getting a head or a tail is identical and equal to 1/2. Thus, the probability of any number in a uniform distribution is given by

$$\mathcal{P}(x) = c$$

where c is a constant. If the numbers are defined on a interval whose length is ℓ then $c = \frac{1}{\ell}$ such that all the probabilities add up to 1.

A *normal* distribution on the other hand produces numbers whose probabilities depend on their value as a Gaussian.

$$\mathcal{P}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

where σ is related to the *variance* of the distribution and μ is the *mean*.

If `rand` or `randn` is called from `Octave` without any arguments, they produce a single number within their domain. Each time you call them, they produce a different number.

```
  octave:1> rand
  ans = 0.56407
  octave:2> rand
  ans = 0.40178
  octave:3> rand
  ans = 0.57570
  octave:4> randn
  ans = 1.3561
  octave:5> randn
  ans = 1.1155
  octave:6> randn
  ans = 0.45327
  octave:7> randn
  ans = -0.89322
```

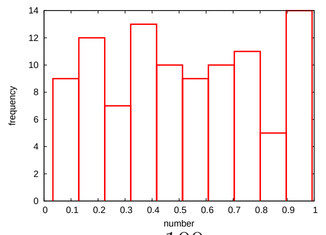
You can also call these two functions with one or two integer arguments, in which case they create a matrix of random numbers with the given distribution.

```
  octave:1> rand(3)
ans =

  0.177001  0.025417  0.749406
  0.678631  0.350609  0.310275
  0.669049  0.770739  0.951104

octave:2> randn(3,2)
ans =

  -1.94691  -0.11158
   0.81064   0.90569
  -1.33847   0.26074

octave:3> rand(4,1)
ans =

  0.16970
  0.25418
  0.34736
  0.70195
```

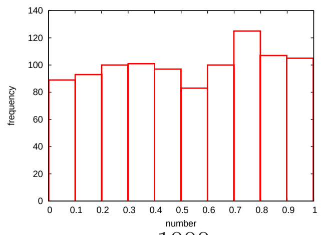
Random number generators usually rely on a semi-deterministic method for generating the desired numbers. To see that `rand` and `randn` indeed yield the desired distribution, let's generate increasingly larger sets of random numbers and let's view their histograms using `hist`.

```
octave:1> r100=rand(100,1);
octave:2> r1000=rand(1000,1);
octave:3> r10000=rand(10000,1);
octave:4> r100000=rand(100000,1);
octave:5> hist(r100,10)
octave:6> hist(r1000,10)
octave:7> hist(r10000,10)
octave:8> hist(r100000,10)
```

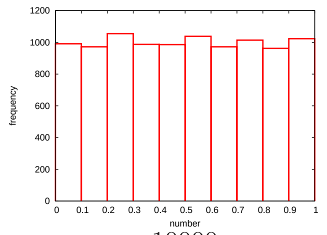
The sequence of plots thus produced looks like the following :



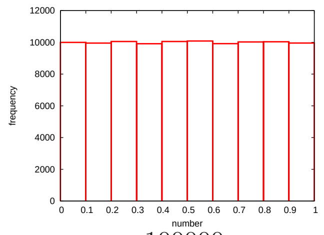
n=100



n=1000

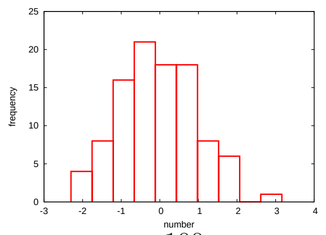


n=10000

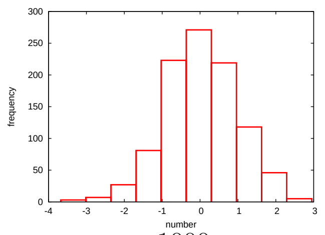


n=100000

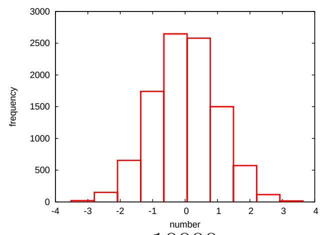
The same procedure may be applied with the `randn` function.



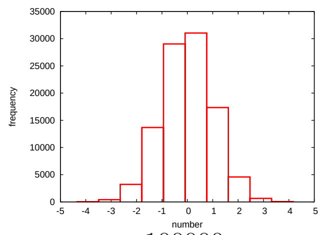
n=100



n=1000



n=10000

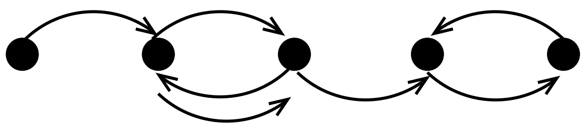


n=100000

As we can imagine, the larger the set of random numbers the more accurate the distribution.

III. RANDOM WALK

Random walk is a simple model to describe motion of particles that undergo random changes of direction during their trajectory. One such particle might be a drunken man who randomly walks around the city picking a random street at every junction. The question is how long it will take for him to reach his house (if ever). This is an example of a two-dimensional random walk on a lattice. An even simpler example is a one dimensional random walk where the *walker* starts at the origin designated by $x_0 = 0$ and takes a step of unit length to the left or to the right according to the results of a coin toss. If it's a head, it moves to the left and if it's a tail it moves to the right. How far away from x_0 does the walker end up after N coin tosses.



First let's analyze this analytically. Let's consider the average displacement of the particle after a very large number of steps, N.

$$\langle x_N \rangle = \sum_i^N s_i$$

where s_i is the displacement at step i. Because we are considering random walk in one dimension, $s_i = \pm 1$. Because for a very large number of steps the walker is just as likely to move to the left as it is to the right, the average total displacement is zero.

A more meaningful property to look at perhaps is the average of the square of the total displacement

$$\langle x_N^2 \rangle = \sum_i^N s_i \sum_j^N s_j$$

For all $i = j$, the second sum yields zero. For $i = j$, then

$$\langle x_N^2 \rangle = \sum_i^N s_i^2 = \sum_i^N 1 = N$$

Thus, the square of the total displacement in an N-step random walk is proportional to N. This is in contrast with a free particle moving with a constant velocity for which the displacement scales like the time. Clearly the random walker moves more slowly. In our simulations, N is synonymous with time. We may generalize the above equation as

$$\langle x^2 \rangle = 2dDt.$$

where D is called the *diffusion constant*, d is the dimensionality of the problem, t is time in seconds and 2 is just a convenient convention.

Let's now prove this by writing a small code. Before writing the code, let's think of some relevant issues :

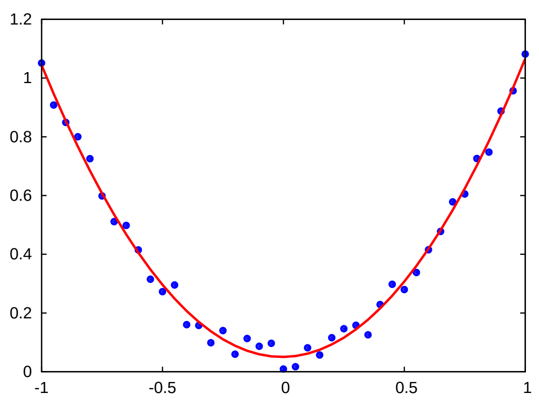
- **The average :** The average $\langle x_N^2 \rangle$ is going to be calculated over N random walks. N random walks will be generated and the square of the cumulative displacement will be calculated. Such and average should be calculated for various N's and collected.
- **Single random walk :** Each random walk produced should be long enough to provide good statistics. On the other hand they shouldn't be so long as to make the simulation prohibitively long.
- **Coin toss :** `Octave` is not able to generate events with discrete outcomes. It can only produce a continuum of numbers with either a uniform or a normal distribution. We can however convert the outcome of the function `rand` to a binary outcome in the following way : for each step of the random walk, we generate a uniformly distributed random number using `rand`. If it's smaller than 0.5, we take a step to the left. If it's larger than 0.5, we take a step to the right.
- **Statistics :** For extremely large N's a $\langle x_N^2 \rangle$-vs-t plot should give us an exactly straight line whose slope is $2D$. However, because our N's are finite, the line will not be exactly straight. In order to get the slope with some statistical error, we'll make a fit to the resulting data using `polyval` and `polyval` in `Octave`

A. polyfit and polyval

In interpreting results from experiments or computer simulations, instead of viewing raw data, we prefer to fit these data to functional forms that are based on some physical intuition. In `Octave` there are two functions that help us do this.

```
octave:1> data=load data-fit;
octave:2> [p,s]=polyfit(data(:,1),data(:,2),2);
octave:3> p
p =

  1.004483  0.010956  0.050520

octave:4> s.normr
ans = 0.18576
octave:5> y=polyval(p,data(:,1));
octave:6> plot(data(:,1),data(:,2),'b*;Raw data;',data(:,1),y,'r-;Fit;')
```

with the outcome



In the above code segment, we notice the following new elements :

1. `load` reads a text file from memory and assigns it to the designated variable, in this case `data`.
2. `polyfit(x,y,n)` does a *least-squares* fit of the arrays x and y to a polynomial of order n. p is an array containing the coefficients of the resulting polynomial and s is a *structure* that contains various matrices and arrays related to the fitting procedure. In particular, the element of `s` called `normr` accessed as demonstrated is a measure of the quality of the fit.
3. `polyval` takes an array as its first argument, another array as its second argument, interprets the first as the coefficients of a polynomial and evaluates the values of this polynomial at the points supplied by the second array. In other words

```
    octave:1> p=[1 2 3]; x=1:100;
    octave:2> y=polyval(p,x);
```

   is equivalent to

```
    octave:3> p(1)*x.^2+p(2)*x+p(3);
```

B. The code for finding the diffusion constant, D

To calculate the average of x^2 over many random walks, we need a loop that creates many random walks, say `Nrw`, of `Nsteps` at every turn and extract x^2 from each. We'll then add up all those values and divide by `Nrw` to find the average. We'll then repeat this for many different `N`'s and plot $\langle x^2 \rangle$ versus N. We'll finally perform the fitting procedure discribed above.

As we gather from the above explanation, we need three *nested* loops in our code. Here, *nested* means one inside the other.

- One loop to generate each random walk, of length `Nsteps`
- One loop to calculate the average of x^2 over many random walks, of length `Nrw`
- And one loop to obtain $\langle x^2 \rangle$ for different N's, of length `Ns`

C. Single random walk in one dimension

To generate a single random walk in one dimension, we need to simulate a coin toss. That is to say, for each step of the random walk, we need create an event which has only to outcomes with equal probability. Such an event might be simulated using `rand` as follows : for each step of the random walk we call `rand`. Now the numbers that are smaller than 0.5 are as likely to come up as numbers that are larger than 0.5. We may thus make a move to the *left* if the number that comes up is *smaller* than 0.5 and to the *right* if it is *larger* than 0.5. We could pack this discrete form of `rand` into a function called, `rand_disc`. Instead of writing a function that returns a single number every time and call it several times from an external loop to create the random walk, it is much more efficient to create all the elements of the random walk at once.

In this section, we'll also see a striking example of how efficient vector operations are in comparison to loops. First let's write a straightforward function that contains a loop in its body :

```
function r=rand_disc_loop(N,num)

rn=[];
for n=1:N

  if (rand<num)
    rn=[rn;-1];
  else
    rn=[rn;1];
    endif

  endfor

endfunction
```

and now let's try to get rid of the loop using specialized array operations coded in `Octave`

```
  function r=rand_disc(N,num)

    rn=rand(N,1);
    r=(rn<num);
    li=find(r == 0);
    r(li)=-1;

  endfunction
```

There are many new concepts that we see in this new function :

- It takes in two arguments : the first one is the obvious length of the random walk and the second is the *offset* of the probability. If we want to create an uneven random walk where the probability of going to the left is *different* from the probability of going to the right, then we could set `num` to be a different number than 0.5. If for example, our walker is three times as likely to go left than right then we could call our function with `num` equal to 0.75.
- The construct

  ```
        r=(rn<num);
  ```

  is a logical assignment. It checks the thruth value of the clause in parentheses for *each* element of the array `rn` and assigns a value of 1 is it's `true` and a value of 0 if it is `false`.
- The `find` statement checks for the logical clause in parantheses next to it and makes a list of those elements of the array that satisfy this statement.

Next, let's compare their efficiency :

```
 octave:1> tic;rand_disc_loop(10000,0.5);toc
 ans = 0.86784
 octave:2> tic;rand_disc(10000,0.5);toc
 ans = 0.0041240
 octave:3> tic;rand_disc_loop(30000);toc
 ans = 8.8272
 octave:4> tic;rand_disc(30000,0.5);toc
 ans = 0.011905
 octave:5> tic;rand_disc_loop(50000,0.5);toc
 ans = 25.911
 octave:6> tic;rand_disc(50000,0.5);toc
 ans = 0.019542
```

`tic` and `toc` are keywords which measure the time (in seconds) it takes to execute the `Octave` commands written inbetween. As you can see the loops take a much longer time to perform the same operation that specialized array operations in `Octave` take. Moreover, this discrepancy grows as the number of operations increase.

D. Putting everything together

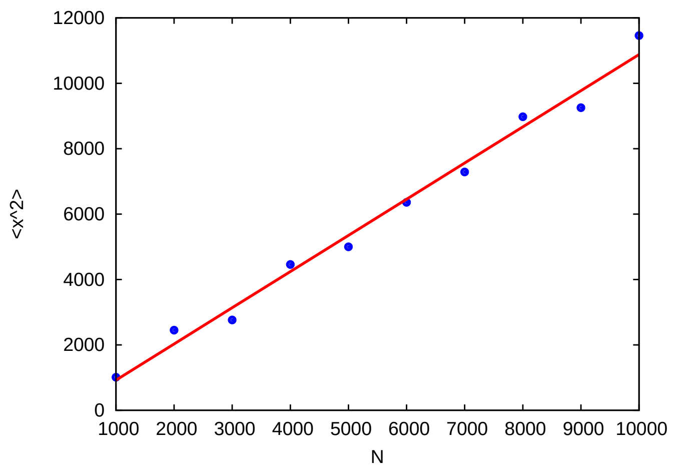
We finally put all of the above pieces of code together and write the following script which we call `random_walk_1d.m`

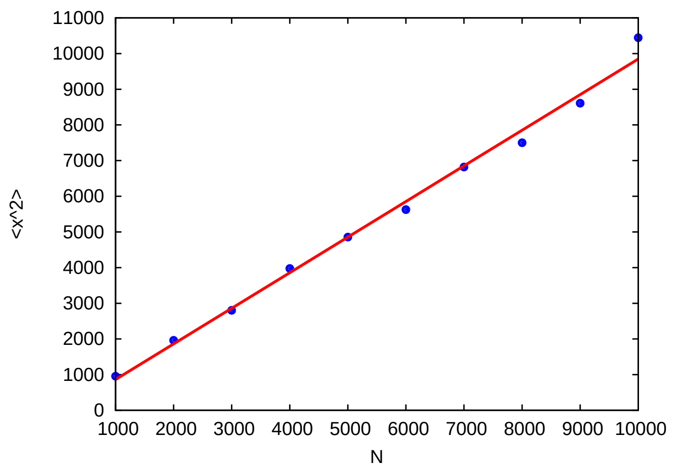
```
## Random walk in one dimension

x2ave=[];
Nrw=3000;
Nstepsmax=10000;
inc=1000;
beg=1000;
dt=1;

for Nsteps=beg:inc:Nstepsmax;
  Nsteps
  x2=0;

  for m=1:Nrw
    r=random_disc(Nsteps,0.5);
    x2+=sum(r)^2;
  endfor

  x2ave=[x2ave;x2/Nrw];

endfor

hold off
plot(dt*[beg:inc:Nstepsmax],x2ave,'b*;;')
[p,s]=polyfit(dt*[beg:inc:Nstepsmax]',x2ave,1);
pval=polyval(p,dt*[beg:inc:Nstepsmax]);
hold on
plot(dt*[beg:inc:Nstepsmax],pval,'r-');
hold off

D=p(1)/2
```

Let's analyze the piece of code above :

1. In the constants section, we set `Nrw` and `Nsteps`. Because we are going to generate random walks with different `Nsteps` values, we set the beginning value, the increment and the final value for `Nsteps`. We also give a value to the time step `dt` which serves only the purpose of given the diffusion constant `D` proper units.
2. `x2ave` is going to be the array of $\langle x^2 \rangle$ values for x^2 over all `Nrw` random walks for different `Nsteps` and therefore we initialize it to an empty array at the beginning.
3. Inside the loop over `Nsteps`, we first display `Nsteps` so that we can watch the algorithm advance. Then we set `x2` to zero since this is the variable that's going to contain the total x^2 from all the random walks.
4. The inner loop over `m` goes over all the random walks and finds the total displacement square for each random walk adding them up. In the end we divide it by the number of random walks we have used and append it to `x2ave`.
5. At the end, we perform the fitting procedure described above.
6. The parts that are written in blue could be turned into functions which would make the above code look tidier and more modular.

Now, let's run this code with two different `Nrw` values, `Nrw=1000` and `Nrw=5000`.





`Nrw`=200
`D`=0.55 units^2/sec
`s.normr`=1131.8

`Nrw`=1000
`D`=0.49 units^2/sec
`s.normr`=791.38

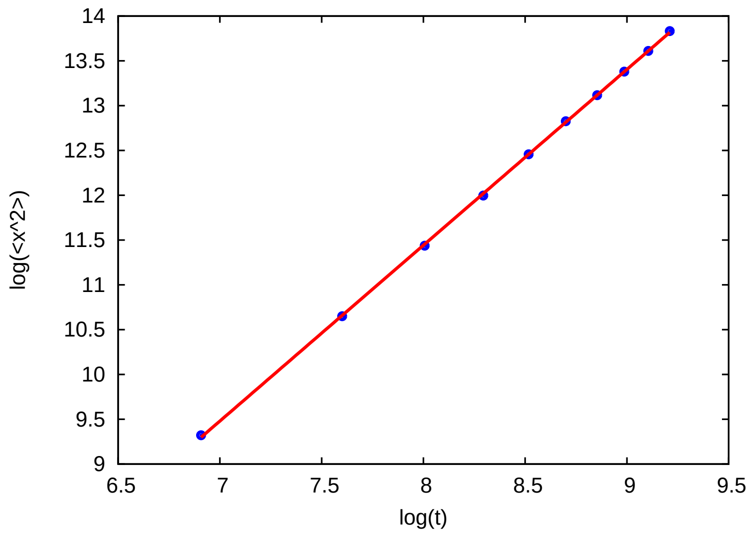
For this simple random walk problem we expect `D` to be 0.5. As you can see from the comparison above, `D` is closer to 0.5 for a larger `Nrw`. This is a reflection of better statistics and it also reflects itself in the quality of the fit given by `s.normr`.

E. Uneven random walk

Let's now run the same code but let's increase the probability of stepping to the right slightly by calling our user-defined function `rand_disc` as `rand_disc(Nsteps,0.45)`. The resulting plot is no longer a straight line. This is because the motion now is somewhere between a real random walk and unhindered motion with constant velocity. We thus expect the average square displacement to go like an exponent of t, which is between 1 and 2. Let's call it α. In order to determine α, we perform the following logarithm trick.

$$\langle x^2 \rangle = Ct^\alpha \quad \Rightarrow \quad \log(\langle x^2 \rangle) = \log C + \alpha \log t$$

If we now make a first-order fit to the logarithms, the slope gives us α.



In the example above, for `num`=0.45, the slope is found to be 1.96. This means that even for a very small offset in the probability, the motion approaches that of a free particle very fast.