Indian Institute of Technology Gandhinagar

PH 102

Simulating Brownian Motion Based on Unpredictability in the Quantum Realm

Team Members:	*Roll Number:*
Kishen N Gowda	17110074
Vraj Patel	17110174
N V Karthikeya	17110090
Nidhin Harilal	17110092
Rushil Shah	17110140
Saumitra Sharma	17110135
Vedanta Krishna Bhutani	17110171

Contents

1 Objective

1. Generating truly random numbers using quantum phenomenon i.e. observing the quantity of electrons tunnelling in emitter reverse biased transistor.
2. Simulating and studying Brownian Motion using the random numbers generated.

2 Theory

2.1 True Random Number Generator:

True Random Number Generator devices generates random numbers by using physical processes, rather than by means of algorithms. Deterministic machines like computers are incapable of generating truly random numbers. Random numbers are of key importance in fields of cryptography and simulating stochastic processes like Brownian motion. Our device is based on amplifying random noise.

This random noise involves the quantum effect of tunneling. We can safely assume Tunneling to be a random phenomenon. This assumption of tunneling being random is also supported by a paper published in nature (Ref. 1). We utilized the avalanche breakdown phenomenon of a common NPN transistor. Any common NPN transistor will fit for job provided its Avalanche breakdown DC voltage value is around 6V (preferably across the base-emitter junction). Although a zener diode can also serve the purpose, following the trend of work done in this area we went with transistors. According to the datasheet of the transistor (2N2222) used, Avalanche breakdown is observed to be the insulator for a reverse bias voltage of 6V around the base-emitter junction. One of the main region to choose Base-emitter junction was that it suits the job, as we are able to generate a potential barrier with the least required reverse voltage (According to the datasheet of transistor collector-emitter junction requires 40V and collector-base requires 75V). We then use again a common NPN junction transistor (this time as amplifier across the base-collector region) to amplify the current obtained. After rectification, we require to convert this random current values to give us a perfect digital output of 5V or (about) 0V. For this, we use a Schmitt trigger.

2.2 Brownian Motion:

Brownian motion (or pedesis), also known as Wiener Process is the random motion of particles suspended in a fluid (a liquid or a gas) resulting from their collision with the fast-moving molecules in the fluid. It is one of the best known and the only continuous path Lévy processes and occurs frequently in pure and applied mathematics, economics, quantitative finance, evolutionary biology, and physics.

In Brownian motion, the particle moves as though under the influence of random forces of varying direction and magnitude.

A standard Wiener process on the interval $[0, T]$ is a random variable $W(t)$ that depends continuously on t in $[0, T]$ and satisfies the following:

$W(0) = 0$
For $0 <= s < t <= T$,
$W(t) - W(s) \sim \sqrt{t-s} \cdot N(0, 1)$,

where $N(0, 1)$ is a normal distribution with zero mean and unit variance. Because the normal distribution is used, the process is often referred to as Gaussian.

For $0 <= s < t < u < v <= T, W(t) - W(s)$ and $W(v) - W(u)$ are independent.

For use on a computer, we discretize the Wiener process with a timestep dt as,

$$\partial W \sim \sqrt{\partial t} N(0, 1).$$

2.2.1 Diffusion Coefficient:

The **diffusion coefficient** is most simply understood as the magnitude of the molar flux through a surface per unit concentration gradient out-of-plane. It is analogous to the property of thermal diffusivity in heat transfer. In an aqueous (water) solution, typical diffusion coefficients are in the range of 10^{-10} to 10^{-9} $m^2 s^{-1}$. The theoretical value of the diffusion coefficient(D) is given by,

$$\mathrm{D{=}k}_B T_{\overline{3\pi\eta d}}$$

where T = temperature (Kelvin), k_B = Boltzmann's constant, η = viscosity, and d = particle diameter.

2.2.2 Scaling the Normal Distribution for Real Particles:

According to theory, the mean squared displacement of the particle is proportional to the time interval,

$$\langle |\vec{r}(t+\tau) - \vec{r}(t)|^2 \rangle = 2dD\tau$$

where r(t) = position, d = number of dimensions, D = diffusion coefficient, and τ = time interval. To generate the correct distribution, the random numbers list (which has a standard normal distribution) must be scaled by the factor k.

2.2.3 Relation between Displacement and Time:

Theory predicts that the displacement should increase in proportion to the square root of time. Since displacement is expected to increase with the square root of time, displacement squared is a straight line. With only a single particle and a small number of samples, deviation from the line can be quite large.

2.3 Correlation Coefficient:

The correlation coefficient is a statistical measure that calculates the strength of the relationship between the relative movements of the two variables. The range of values for the correlation coefficient bounded by 1.0 on an absolute value basis or between -1.0 to 1.0. If the correlation coefficient is greater than 1.0 or less than -1.0, the correlation measurement is incorrect. A correlation of -1.0 shows a perfect negative correlation, while a correlation of 1.0 shows a perfect positive correlation. A correlation of 0.0 shows zero or no relationship between the movement of the two variables.
While the correlation coefficient measures a degree of relation between two variables, it only measures the linear relationship between the variables. The correlation coefficient cannot capture nonlinear relationships between two variables.
A value of exactly 1.0 means there is a perfect positive relationship between the two variables. For a positive increase in one variable, there is also a positive increase in the second variable. A value of -1.0 means there is a perfect negative relationship between the two variables. This shows the variables move in opposite directions — for a positive increase in one variable, there is a decrease in the second variable. If the correlation is 0, there is no relationship between the two variables.
The strength of the relationship varies in degree based on the value of the correlation coefficient. For example, a value of 0.2 shows there is a positive relationship between the two variables, but it is weak and likely insignificant. Experts do not consider correlations significant until the value surpasses at least 0.8. However, a correlation coefficient with an absolute value of 0.9 or greater would represent a very strong relationship. As Brownian Motion are stochastic processes, the correlation coefficient of any two Brownian motion must be close to zero.

The Correlation Coefficient of two variables is given by,

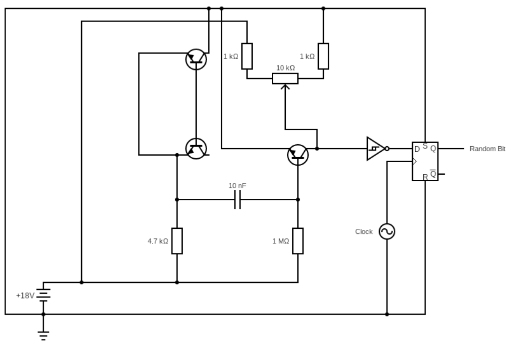
$$r = \frac{C_{ov}(X,Y)}{\sigma_X \sigma_Y}$$

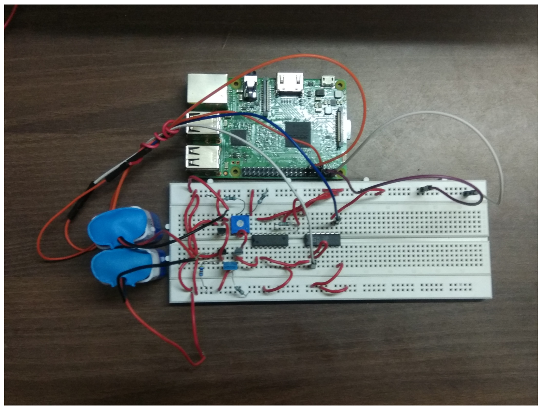
$$r = \frac{\sum_1^N (X_i - \overline{X})(Y_i - \overline{Y})}{\sigma_X \sigma_Y (N-1)}$$

3 Materials Required

1. Transistors (2N2222)
2. Inverting Schmitt Trigger NOT gate
3. Potentiometer (10 K ohm)
4. Raspberry Pi
5. 18 V Supply (9 V Batteries or DC power supply)
6. Resistors (See circuit)
7. Capacitors (See circuit)
8. D flip-flop
9. LEDs

4 Experimental Setup





5 Part 1

5.1 Procedure

1. Make the circuit as given in the above circuit diagram. Since we are making a high-gain amplifier in the circuit, make all the connections as short as possible to avoid picking up stray electromagnetic fields from the surrounding.

2. Try to place all transistors and capacitors as close as possible. Cut the unused collector lead of the noise-generating transistor (A).
3. Give a 90% duty cycle clock of 1 Hz to the flip-flop and the 5V power supply to ICs through Raspberry Pi using the code given:
4. Rotate the potentiometer until you get a good amount of blinking in the LED.
5. Now remove the LED. Now take the input into the Raspberry Pi GPIO.
6. Use the code given below, making appropriate changes based on the pin numbers used.
7. Make appropriate changes for the number of bits required and you get a file made up of the random bits. Repeat the runs 3 times generating 4,00,000 bits on each run and get a set of 3 files.
8. Further analysis is carried out on these files to simulate Brownian Motion.

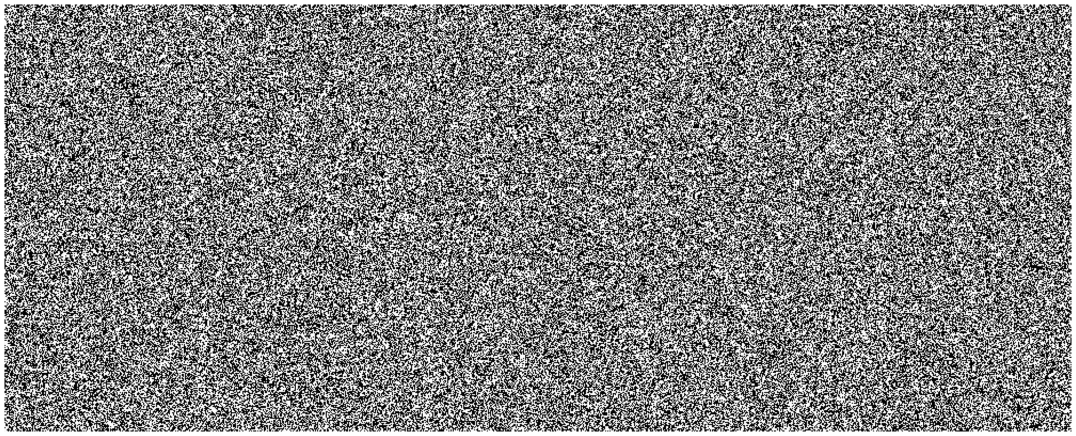
Code:

```
import RPi.GPIO as GPIO

from time import sleep

GPIO.setmode(GPIO.BOARD)
GPIO.setup(7,GPIO.OUT)
GPIO.setup(11,GPIO.IN)
freq = 1000
T = 1/freq
p = GPIO.PWM(7,freq)
p.start(0.9)
num = 1
nbits = 2000000
with open('randbits2.txt','wb') as file:
    for i in range(num):
        print(i+1)
        numstr = ''
        j = 0
        while(j<nbits):
            sleep(T)
            b1=GPIO.input(11)
            sleep(T)
            b2=GPIO.input(11)
            if(b1+b2 == 1):
                j+=1
                numstr += str(b1)
                if(j%32== 0):
```

```
                print(j) if j%1024==0 else 0
                file.write(int(numstr[::-1],2).to_bytes(4,'little'))
                numstr = ''
        #file.write(str(int(numstr,2))+'\n')
p.stop()
GPIO.cleanup()
```

5.2 Analysis & Results

Run Number	Arithmetic Mean Bytewise	Monte Carlo Pi Value	% Error in Pi	Chi-Squared % Test	Number of 0s	Number of 1s
1	127.205	3.163326533	0.69	25.84	199803	200197
2	127.8013	3.147965919	0.2	25.36	199315	200685
3	127.6148	3.155166207	0.43	30.93	199336	200664



Representation of Random Noise Generator
Black Pixel = 1, White Pixel = 0

5.3 Errors

- The source is assumed to maintain a uniform probability of giving ones and zeroes over time. Therefore, while de-skewing, we will not get perfect uniform distribution.
- Variation in the surroundings can disrupt uniform distribution.
- Stray electromagnetic fields might be caught and current values amplified by the amplifier.

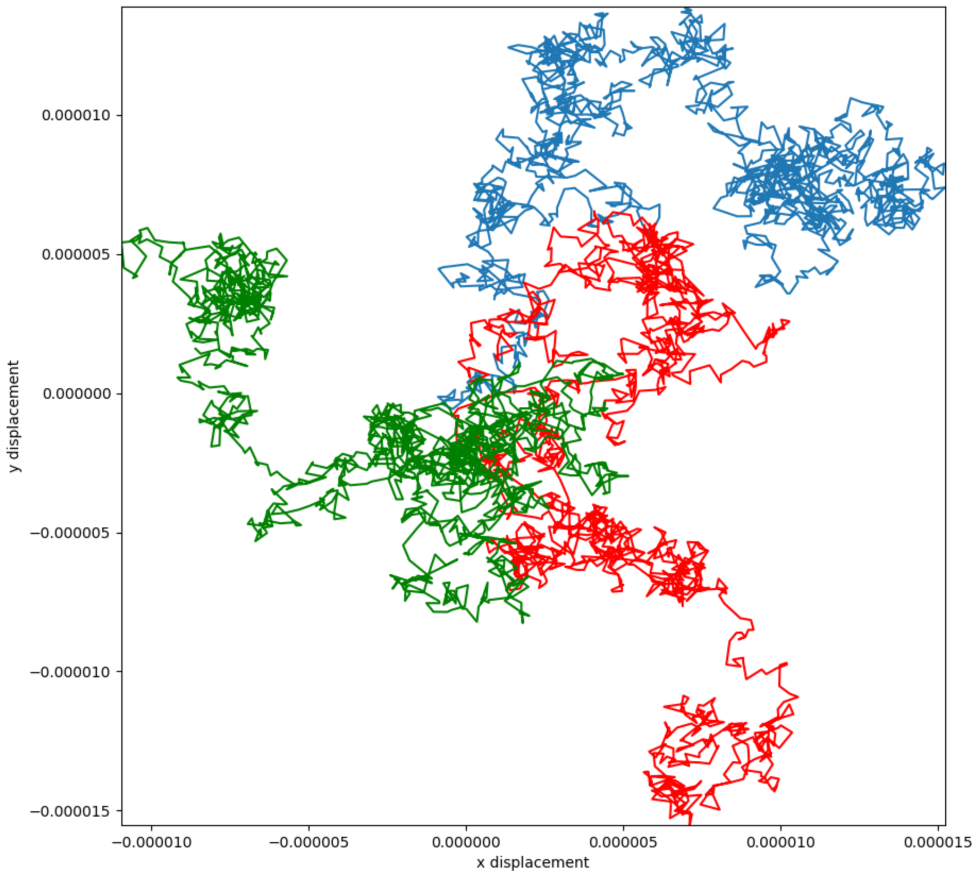
6 Part 2

We will use the random data obtained and now we will simulate and study some properties of Brownian motion. The code is given in the Appendix for reference.

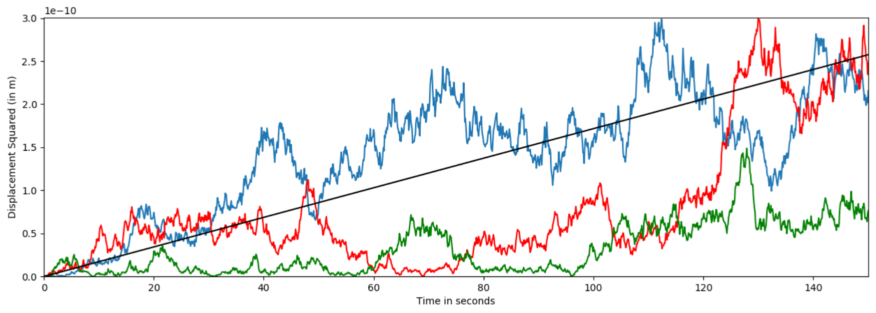
6.1 Procedure

1. Take 8 bits at a time and treat it as a single integer.
2. We get a uniform distribution of numbers from 0 to 255.
3. Convert these numbers to a normal distribution using the Box-Muller method.
4. Now we have a list of random numbers following a normal distribution.
5. Calculate the diffusion coefficient you desire for any particular fluid.
6. Now simulate a Wiener process which is a one-dimensional Brownian Motion but scale the displacements based on the diffusion coefficient calculated.
7. Now we take one Wiener process as the X-axis displacement and other as Y-axis displacement.
8. Now, we plot this and get the Brownian Motion Path
9. Also plot the Displacement Squared v/s time graph.
10. Calculate the diffusion coefficient you desire from the simulation using the formula, in theory.

6.2 Analysis & Results



Brownian Motion



Displacement^2 $v/s\,Time$

6.3 Calculations and Error Analysis:

6.3.1 Estimated Diffusivity from simulated data:

From theory,
simulatedD $= \frac{mean(dSquaredDisplacement)}{2 \times dimensions \times \tau}$
From simulated data,
$D_1 = 4.197943082912188 \times 10^{-13} m^2 s^{-1}$
$D_2 = 4.0920671801477695 \times 10^{-13} m^2 s^{-1}$
$D_3 = 4.4991802113610167 \times 10^{-13} m^2 s^{-1}$
Therefore, simulatedD $= \frac{D1+D2+D3}{3} = 4.263063491473658 \times 10^{-13} m^2 s^{-1}$
Error Analysis:
standardError $= \frac{\sigma(dSquaredDisplacement)}{2 \times dimensions \times \tau * sqrt(N)}$
actualError $=$ D - simulatedD

From Simulation data:
$Err_1 = 1.0669802737044507 \times 10^{-14} m^2 s^{-1}$
$Err_2 = 1.0429709260666482 \times 10^{-14} m^2 s^{-1}$
$Err_3 = 1.1470266441194321 \times 10^{-14} m^2 s^{-1}$
standardError $= \frac{err1+err2+err3}{3} = 1.0856592812968437\ X\ 10^{-14} m^2 s^{-1}$
actualError $= 2.71171545 \times 10^{-15} m^2 s^{-1}$

6.3.2 Correlation Coefficient of two stochastic Processes:

From simulation data,
$r_1 = 0.32$ (Between W1 and W2)
$r_2 = 0.54$ (Between W1 and W3)
$r_3 = 0.53$ (Between W2 and W3)

7 Discussion/Concluding Remarks

7.1 PART I

For an ideal random sequence of large length expectation of the various quantities is as follows,

- Arithmetic Mean Bytewise: 127.5
- Monte Carlo Pi : as close to real Pi as possible
- Chi-Squared Test %: should lie between 10% and 90%

- Number of zeroes should be close to the number of ones

We observe that the data obtained after running various tests for randomness says that the sequence that we obtained is indeed random.

7.2 PART II

From the Calculations, we can observe that,
The original value of D (Coefficient of Diffusion) = $4.2901806459851305 \times 10^{-13} m^2 s^{-1}$
Calculated Value of D,
simulatedD = $4.263063491473658 \times 10^{-13} m^2 s^{-1}$
Standard Error = $1.0856592812968437 \times 10^{-14} m^2 s^{-1}$
actualError = $2.71171545 \times 10^{-15} m^2 s^{-1}$

Also,
The Calculated Coefficient of Correlation,
$r_1 = 0.32$
$r_2 = 0.54$
$r_3 = 0.53$
As we can notice that the values are between 0 and 0.8 which means that there is less correlation between every pair of Weiner processes.

8 References

- https://me.ucsb.edu/ moehlis/APC591/tutorials/tutorial7/node2.html
- https://lben.epfl.ch/files/content/sites/lben/files/users/179705/Simulating%20Brownian%20Motion.
- https://www.comsol.com/multiphysics/diffusion-coefficient
- https://www.investopedia.com/terms/c/correlationcoefficient.asp

9 Appendix

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.animation as animation
from math import pi,sqrt,sin,cos,log, ceil
two_pi = 2*pi

def wiengen(randlist,time_step,N,D):
        k = sqrt(2*D*time_step)
        dt = time_step
        dW = []
        W = []
        dW.append(randlist.pop() * k)
        W.append(dW[0])
        for i in range(1,N):
                dW.append( randlist.pop() *k)
                W.append(W[-1] + dW[-1])
        return W, dW

def bmmethod(l):
        rl = []
        MAXRAND = max(l)
        while l:
                u1 = l.pop() * (1.0/MAXRAND)
                u2 = l.pop() * (1.0/MAXRAND)
                while(not u1):
                        u1 = l.pop() * (1.0/MAXRAND)
                        u2 = l.pop() * (1.0/MAXRAND)
                z1 = sqrt(-2.0 * log(u1)) * cos(two_pi * u2)
                z2 = sqrt(-2.0 * log(u1)) * sin(two_pi * u2)
                rl.append(z1)
                rl.append(z2)
        return rl
binstr = lambda x: bin(x)[2:]
for i in range(len(l)//2)]
def file_get(fname):
        s = open(fname,"rb")
        k = [int(i) for i in s.read()]
        s.close()
        return k
randlist = bmmethod(file_get('randbits'))
randlist2 = bmmethod(file_get('randbits2'))
randlist3 = bmmethod(file_get('randbits3'))
d = 10**(-6) #Diameter of Particle
eta = 10**(-3) #Viscosity of Fluid
kB = 1.38 * (10**(-23)) #Boltzmann constant
Temp = 293 #Temparature in Kelvin
D = kB * Temp/(3*pi*eta*d) #diffusion coefficient
#print(D)
time_step = 0.1
time_total = 150
N = ceil(time_total/time_step)
```

```
#Weiner process
W1, dW1 = wiengen(randlist[:N+2],time_step,N,D)
W2, dW2 = wiengen(randlist[N:],time_step,N,D)

W1_1, dW1_1 = wiengen(randlist2[:N+2],time_step,N,D)
W2_1, dW2_1 = wiengen(randlist2[N:],time_step,N,D)

W1_2, dW1_2 = wiengen(randlist3[:N+2],time_step,N,D)
W2_2, dW2_2 = wiengen(randlist3[N:],time_step,N,D)

## Squared Displacement
squaredDisplacement = np.array(W1)**2 + np.array(W2)**2
dsquaredDisplacement = np.array(dW1)**2 + np.array(dW2)**2
squaredDisplacement_1 = np.array(W1_1)**2 + np.array(W2_1)**2
dsquaredDisplacement_1 = np.array(dW1_1)**2 + np.array(dW2_1)**2
squaredDisplacement_2 = np.array(W1_2)**2 + np.array(W2_2)**2
dsquaredDisplacement_2 = np.array(dW1_2)**2 + np.array(dW2_2)**2

##Estimating Coefficient of Diffusivity

Diff = np.mean(dsquaredDisplacement)/(2*2*time_step)
#print(Diff)
Diff_1 = np.mean(dsquaredDisplacement_1)/(2*2*time_step)
Diff_2 = np.mean(dsquaredDisplacement_2)/(2*2*time_step)
#print("D = " + str(D),"Diff = " +  str(Diff),"Diff_1 = " + str(Diff_1),"Diff_2 = " +  s

##Error Analysis
err = np.std(dsquaredDisplacement)/(2*2*time_step*sqrt(N))
err_1 = np.std(dsquaredDisplacement_1)/(2*2*time_step*sqrt(N))
err_2 = np.std(dsquaredDisplacement_2)/(2*2*time_step*sqrt(N))
#print(err, err_1, err_2, (err+err_2+err_1)/3)
##Co-relation Coefficient
# Displacement List
w1 = np.sqrt(np.array(W1)**2 + np.array(W2)**2)
w2 = np.sqrt(np.array(W1_1)**2 + np.array(W2_1)**2)
w3 = np.sqrt(np.array(W1_2)**2 + np.array(W2_2)**2)

#Coefficient of Correlation
r1 = np.dot(w1 - np.mean(w1),w2 - np.mean(w2))/(N-1)
r2 = np.dot(w1 - np.mean(w1),w3 - np.mean(w3))/(N-1)
r3 = np.dot(w2 - np.mean(w2),w3 - np.mean(w3))/(N-1)

#print('r1 = ' +str(r1) + '\nr2 = ' +str(r2) + '\nr3 = ' +str(r3))
```

```
##Animation
MAP = 'plasma'
fig = plt.figure()
lim = lambda q: (min(q)-0.5,max(q)+0.5)
ax = plt.axes(xlim=lim(W1), ylim=lim(W2))
cm = plt.get_cmap(MAP)]
ax.set_prop_cycle(color= cmc))
cmc = [cm(1.*i/(N-1)) for i in range(N-1
line, = ax.plot(np.array([0,0]),np.array([0,0]), lw=1,animated=True)

x = [np.nan]*N
y = [np.nan]*N

def init():
        line, = ax.plot(x,y,lw=1,animated=True)
        return line,

def animate(i):
        global x
        global y
        x[i]=W1[i]
        y[i]=W2[i]
        if i>=1:
                line, = ax.plot([x[i-1],x[i]],[y[i-1],y[i]],lw=1)
        else:
                line, = ax.plot(np.array([0,0]),np.array([0,0]),lw=1)
        return line,

anim = animation.FuncAnimation(fig,animate,init_func=init,frames = N-1,interval = 1,blit
plt.show()
plt.rcParams["figure.figsize"] = (10,10)
fig = plt.figure()
lim = lambda q: (min(q),max(q))
ax = plt.axes(xlim=lim(W1+W1_1+W1_2), ylim=lim(W2+W2_1+W2_2))
plt.plot(W1,W2)
plt.plot(W1_1,W2_1,c='r')
plt.plot(W1_2,W2_2,c='g')
plt.xlabel("x displacement")
plt.ylabel("y displacement")
plt.savefig('abc.png')
plt.show()

##Displacement squared vs time graph
disp = lambda x,y: (x**2 + y**2)
```

```
darr = [disp(W1[i],W2[i]) for i in range(N)]
darr2 = [disp(W1_1[i],W2_1[i]) for i in range(N)]
darr3 = [disp(W1_2[i],W2_2[i]) for i in range(N)]
x_theoretical = [i*time_step for i in range(1,N+1)]
print(D)
y_theoretical = [i*4*D*time_step for i in range(1,N+1)]
plt.axes(xlim=(0,time_total),ylim = (min(*y_theoretical,*darr,*darr2,*darr3),max(*y_theo
plt.plot([i*time_step for i in range(1,N+1)],darr)
plt.plot([i*time_step for i in range(1,N+1)],darr2,c='r')
plt.plot([i*time_step for i in range(1,N+1)],darr3,c='g')
plt.plot(x_theoretical,y_theoretical,color='k')
plt.xlabel("Time in seconds")
plt.ylabel("Displacement Squared (in m)")
plt.show()
```