Exam 1

Ryan Branagan

February 12, 2019

1 Charges on a Line

```
#Ryan Branagan
#Collaborators: Jack Featherstone, Grant Sherrill, JP Habeeb, Emma Stone, Crosson Nipper
#Branagan_ex1_p1.py
#2/9/19

import numpy as np
import pylab as p

#Parameters
n = 200
e = 3

#Define the potential as a funciton of N
def P(N):
    P = np.array([])
    for i in range(1,N+1):
        if i % 2 == 0:
            T = 1
        elif not i % 2 == 0:
            T = -1

        P = np.append(P,((2*T)/i))
    return np.sum(P)

#Input list
Nlist = np.linspace(1,n,n)

#Output list
ans = np.array([])
for A in Nlist:
```

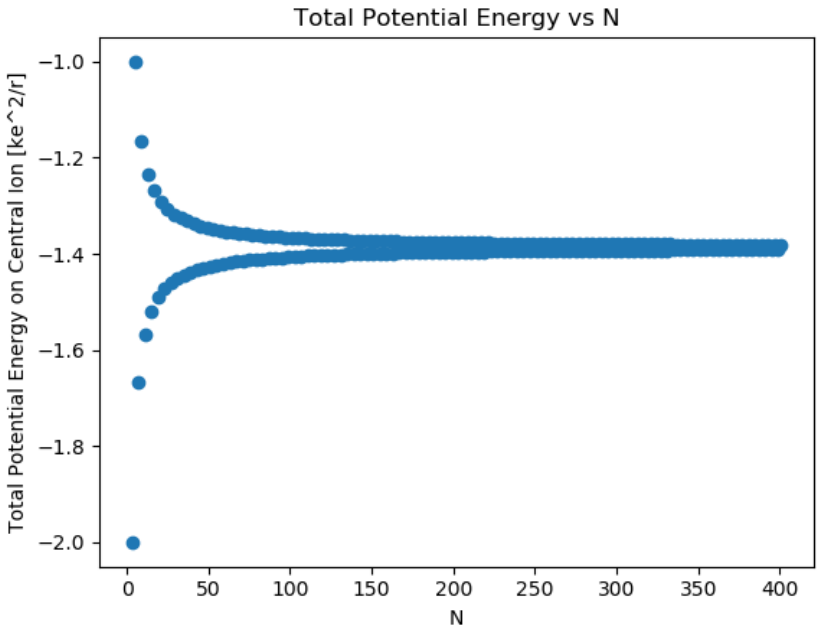


Figure 1: I cannot tell exactly what value it converges to but it is slightly greater than -1.4.

```
    ans = np.append(ans,P(int(A)))

x = 2*Nlist+1

#Plotting
fig, ax = p.subplots(1,1)
ax.plot(x,ans,'o')
ax.set_xlabel('N')
ax.set_ylabel('Total Potential Energy on Central Ion [ke^2/r]')
ax.set_title('Total Potential Energy vs N')
p.show()
```

To solve this problem I started with about 9 ions and worked it out by hand. I started by assuming the central atom was positive. After drawing out a few ions, I noticed that there is symmetry about the central ion. Taking advantage of this, I decided to make ions only extend on one side and multiplied by two. From there, keeping in units of $\frac{ke^2}{r}$, the potential energy becomes the fraction of the sign of the charge times two over the multiple of r. With this I defined a function that when given a number N coming from $2 * N + 1$ returns the total potential energy of the first atom. Next, I used multiple values of N and plotted my results.

2 Position, Velocity, Acceleration

```
#Ryan Branagan
#Collaborators: Jack Featherstone, Grant Sherrill, JP Habeeb, Emma Stone, Crosson Nipper
#Branagan_ex1_p2.py
#2/9/19

import numpy as np
import pylab as p

#Parameters
a = -2
b = 2
points = 1000

#Part a
def func(t):
    if np.abs(t) < 1:
        return np.exp(-2*((np.arctanh(t))**2))
    else:
        return 0

ts = np.linspace(a,b,points)

ans = np.array([])
for t in ts:
    ans = np.append(ans,func(t))

fig1, ax = p.subplots(1,1)
ax.plot(ts,ans,'o')
ax.set_xlabel('t')
ax.set_ylabel('f(t)')
ax.set_title('f(t) vs t')
p.show()
#%%
#Part B

#Define Integration Technique
#I don't need Simpson's Rule but here it is
def simp3(f,a,b,N):
    h = (b-a)/N

    #Left
    xl = np.array([])
```

```
    for A in range(N):
        xl = np.append(xl,(a+A*h))
    Left = np.sum(f(xl)*h)

    #Right
    xr = np.array([])
    for B in range(N):
        xr = np.append(xr,(a+(B+1)*h))
    Right = np.sum(f(xr)*h)

    #Trap
    Trap = (Left+Right)/2

    #Mid
    xm = np.array([])
    for C in range(N):
        xm = np.append(xm,(a+(C+1/2)*h))
    Mid = np.sum(f(xm)*h)

    return (Trap/3)+(2*Mid/3)

def DiffEqs(f,a,b,dt,strr):
    if strr == 'No':
        alist = np.array([f(a)])
        blist = np.array([0])
        ts = np.linspace(a,b,(1/dt))
        for i,t in enumerate(ts):
            y = alist[i]+f(t)*dt
            alist = np.append(alist,y)
            z = blist[i]+alist[i]*dt
            blist = np.append(blist,z)
        alist = np.delete(alist,len(ts))
        blist = np.delete(blist,len(ts))
        return ts, alist, blist
    if strr == 'Yes':
        alist = np.array([f(a)])
        ts = np.linspace(a,b,(1/dt))
        for i,t in enumerate(ts):
            y = alist[i]+f(t)*dt
            alist = np.append(alist,y)
        alist = np.delete(alist,len(ts))
        return ts, alist

#Define Derivative Technique
```

```
def D5PS(f,x,h):
    return (f(x-2*h)-8*f(x-h)+8*f(x+h)-f(x+2*h))/(12*h)

def D23PS(f,x,h):
    return (f(x+h)-2*f(x)+f(x-h))/(h**2)
#%%
#Case 1
f = func
a = -2
b = 2
h = 10**-3

#T values
ts = np.linspace(a,b,(1/h))

#Getting points
position = np.array([])
velocity = np.array([])
acceleration = np.array([])
for t in ts:
    velocity = np.append(velocity,D5PS(func,t,h))
    acceleration = np.append(acceleration,D23PS(func,t,h))
    position = np.append(position,func(t))

#Plotting
fig2, (pos,vel,accel) = p.subplots(3,1)

pos.plot(ts,position,'o')
pos.axhline(0, color='k', linestyle='--')
pos.set_ylabel('Position [m]')
pos.set_title('Case 1: Position, Velocity, Acceleration vs Time')

vel.plot(ts,velocity,'o')
vel.set_ylabel('Velocity [m/s]')
vel.axhline(0, color='k', linestyle='--')

accel.plot(ts,acceleration,'o')
accel.set_ylabel('Acceleration [m/s^2]')
accel.set_xlabel('Time [s]')
accel.axhline(0, color='k', linestyle='--')

p.show()
#%%
#Case 2
```

```
f = func
a = -2
b = 2
dt = 10**-3
strr = 'Yes'

#Getting points
ts,position = DiffEqs(f,a,b,dt,strr)

velocity = np.array([])
acceleration = np.array([])
for t in ts:
    acceleration = np.append(acceleration,D5PS(f,t,dt))
    velocity = np.append(velocity,func(t))

#Plotting
fig3, (pos,vel,accel) = p.subplots(3,1)

pos.plot(ts,position,'o')
pos.axhline(0, color='k', linestyle='--')
pos.set_ylabel('Position [m]')
pos.set_title('Case 2: Position, Velocity, Acceleration vs Time')

vel.plot(ts,velocity,'o')
vel.set_ylabel('Velocity [m/s]')
vel.axhline(0, color='k', linestyle='--')

accel.plot(ts,acceleration,'o')
accel.set_ylabel('Acceleration [m/s^2]')
accel.set_xlabel('Time [s]')
accel.axhline(0, color='k', linestyle='--')

p.show()
#%%
#Case 3
f = func
a = -2
b = 2
dt = 10**-3
strr = "No"

#Getting points
ts,velocity,position = DiffEqs(f,a,b,dt,strr)
```

```
acceleration = np.array([])
for t in ts:
    acceleration = np.append(acceleration,func(t))

#Plotting
fig4, (pos,vel,accel) = p.subplots(3,1)

pos.plot(ts,position,'o')
pos.axhline(0, color='k', linestyle='--')
pos.set_ylabel('Position [m]')
pos.set_title('Case 3: Position, Velocity, Acceleration vs Time')

vel.plot(ts,velocity,'o')
vel.set_ylabel('Velocity [m/s]')
vel.axhline(0, color='k', linestyle='--')

accel.plot(ts,acceleration,'o')
accel.set_ylabel('Acceleration [m/s^2]')
accel.set_xlabel('Time [s]')
accel.axhline(0, color='k', linestyle='--')

p.show()
```

Part a was very straight forward. I created a definition based on the given function, then graphed it. Part b was not very straight forward. In theory, you can just take the derivative or integral to solve this problem. But the whole point of going over numerical derivatives and integrals is to deal with nasty functions like the one given. So I started off by picking Simpson's rule, I thought this would be a good integration technique and rewrote it as its own definition instead of using other definitions to define it. After spending a good amount of time doing this I realized I could not use it because I could not numerically integrate twice. Later, I also realized that this would not work because the value of a function at a point is fundamentally different from the integral which is the area under the curve. However, a while after realizing this, my classmates showed that this method does work. The shape of the graph is the same but the values are not. Ultimately, my choice of integration technique was Euler's Method. This would give my points that I could graph and I could easily solve for the velocity and position at the same time for the third case. I ran into problems coding this because I used an appending method which gave me one more value in my output arrays than needed. So, I just deleted the last value since I could not change the enumerate to give me one less index value. The derivatives were not as much work since we already had to be very familiar with the first and second derivatives for our homework. Since we had used them on the homework I used a centered five point stencil for the first derivative and a centered three point stencil for the second derivative. Applying all these techniques appropriately I obtained graphs for the position, velocity, and acceleration in each of the

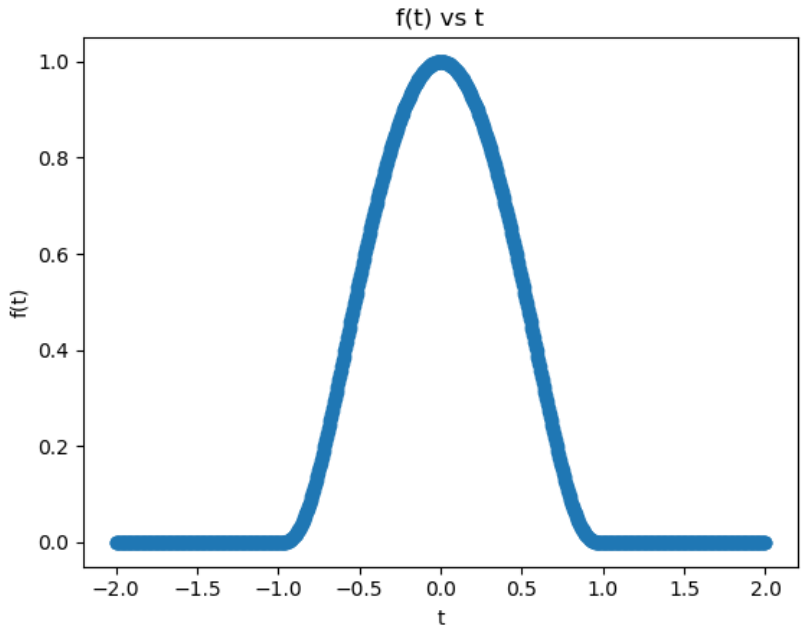
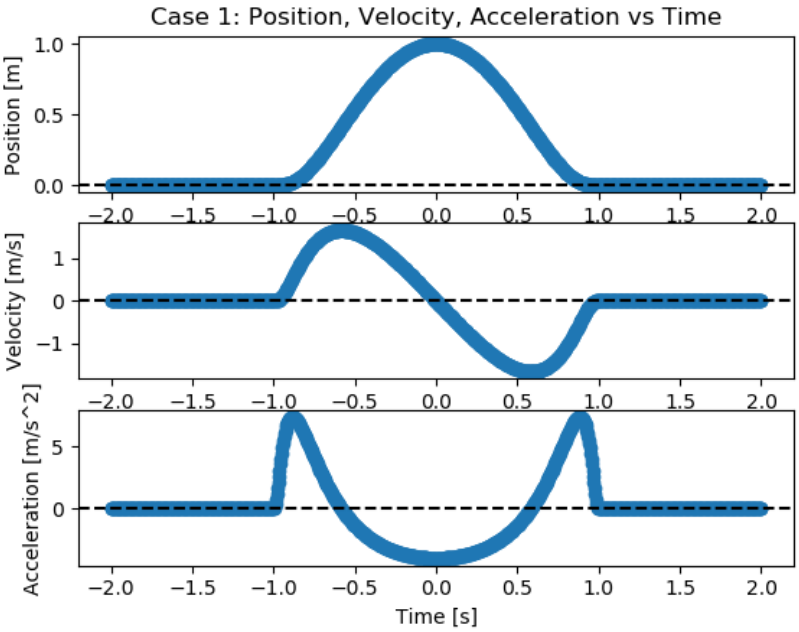


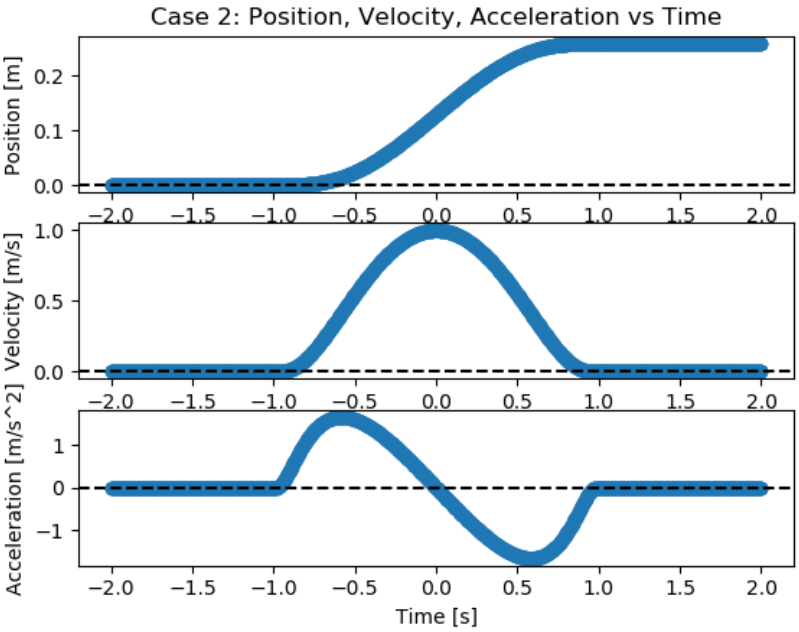
Figure 2: Part a

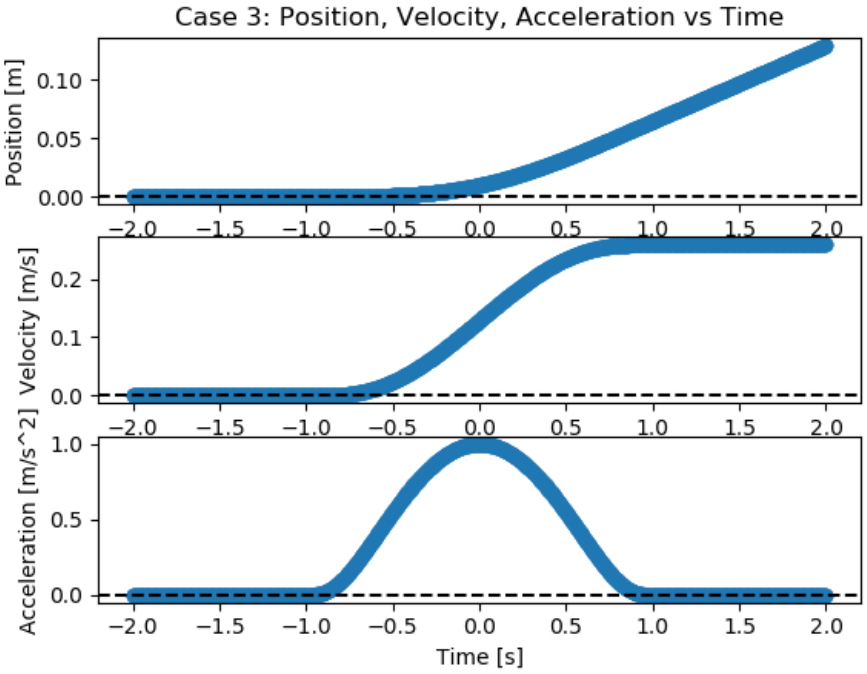
three cases.

3 Road to Nowhere

```
#Ryan Branagan
#Collaborators: Jack Featherstone, Grant Sherrill, JP Habeeb, Emma Stone, Crosson Nipper
#Branagan_ex1_p3.py
#2/9/19

import numpy as np
import pylab as p

#%%
#Part a
displacement = np.array([])
y = 0
for i in range(100):
    y = y + np.random.choice([-1,1])
    displacement = np.append(displacement,y)

ts = np.linspace(0,99,100)
```


Case 1: Position, Velocity, Acceleration vs Time
Position [m]
Velocity [m/s]
Acceleration [m/s^2]
Time [s]


Case 2: Position, Velocity, Acceleration vs Time
Position [m]
Velocity [m/s]
Acceleration [m/s^2]
Time [s]




```
fig1, dis = p.subplots(1,1)
dis.plot(ts,displacement)
dis.set_xlabel('Time/Step Number')
dis.set_ylabel('Distance from Origin')
dis.set_title('Distance from Origin vs Time')
p.show()
#%%
#Part b
N = 8000

def FDisp():
    A = np.array([])
    for i in range(100):
        z = np.random.choice([-1,1])
        A = np.append(A,z)
    return np.sum(A)

FDispList = np.array([])
for i in range(N):
    FDispList = np.append(FDispList,(FDisp()))
```

```
Avg = np.abs(np.sum(FDispList)/N)
print(Avg)

p.figure(2)
p.hist(FDispList,20)
p.xlabel('Final Displacement')
p.ylabel('Number of Times Occured')
p.title('Frequency of Certain Final Displacements')
p.show()
#%%
#Part c
N = 8000

FDispListS = np.array([])
for i in range(N):
    FDispListS = np.append(FDispListS,(FDisp())**2)

RMS = np.sqrt(np.sum(FDispListS)/N)
print(RMS)

p.figure(3)
p.hist(FDispListS,20)
p.xlabel('Final Displacement Squared')
p.ylabel('Number of Times Occured')
p.title('Frequency of Certain Final Displacements Squared')
p.show()
#%%
#Part d
Trials = 50

def FDisp2(S):
    A = np.array([])
    for i in range(S):
        z = np.random.choice([-1,1])
        A = np.append(A,z)
    return np.sum(A)

steplist = np.linspace(10,5000,50)

Step1 = np.array([])
Step2 = np.array([])
for s in steplist:
    for i in range(Trials):
        Step1 = np.append(Step1,(FDisp2(int(s)))**2)
```

```
    Step2 = np.append(Step2,(np.sqrt(np.sum(Step1)/Trials)))
    Step1 = np.array([])

fig4, (p1,p2) = p.subplots(2,1,figsize=(6,8))

p1.plot(steplist,Step2,'o')
p1.set_xlabel('Steps')
p1.set_ylabel('RMS Displacment')
p1.set_title('RMS Displacement vs Steps')

p2.plot(np.log10(steplist),np.log10(Step2),'o')
p2.set_xlabel('Log of Steps')
p2.set_ylabel('Log of RMS Displacement')
p.show()
```

Very near me starting the problem, I got help from a classmate telling me about random.choice instead of random.randint which would give 0s which is not an option for R1D1. At first, I didn't read the problem correctly and calculated the final displacement after 100 steps, this would be useful for all the other parts but not part a. After, figuring out I didn't properly answer the question, I calculated R1D1's distance from the origin at a given time by using a method similar to adding the first N integers or adding the first N squares. I then graphed the output. For part b I used my calculation of the final displacement to answer the question. The way I calculated the final displacement was by collecting each movement in an array and then summing up the whole array. In this way I was able to cut down on the number of calculations so that running 8000 trials was relatively fast. My expected value of the average of R1D1's displacement was 0.1445. This is a value very close to zero which is what we expect, on average R1D1 should have zero displacement from the origin. This value in particular is poor compared to other values I have gotten when running my code before. My process for part c was very similar to part b, except the obvious calculating the square of the final displacement and finding the RMS displacement. My expected value this time was about 9.88. At first this seems odd but if you just look at the final displacement for a few iterations you will see that in general the absolute values average to about 10. Lastly, in part d I defined the final displacement in terms of number of steps then used a nested for loop to get a list of RMS values for varying number of steps. Plotting my results I made a linear and log-log plot.

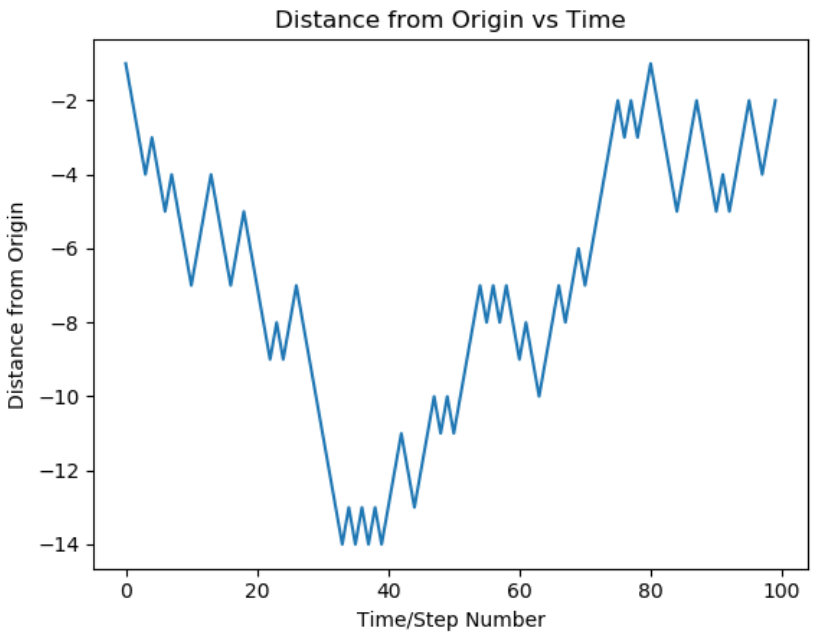
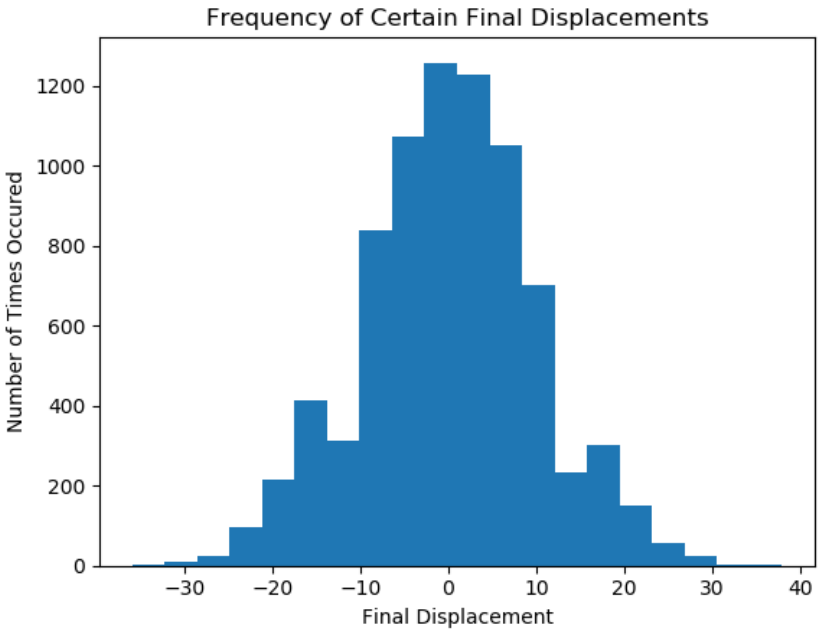
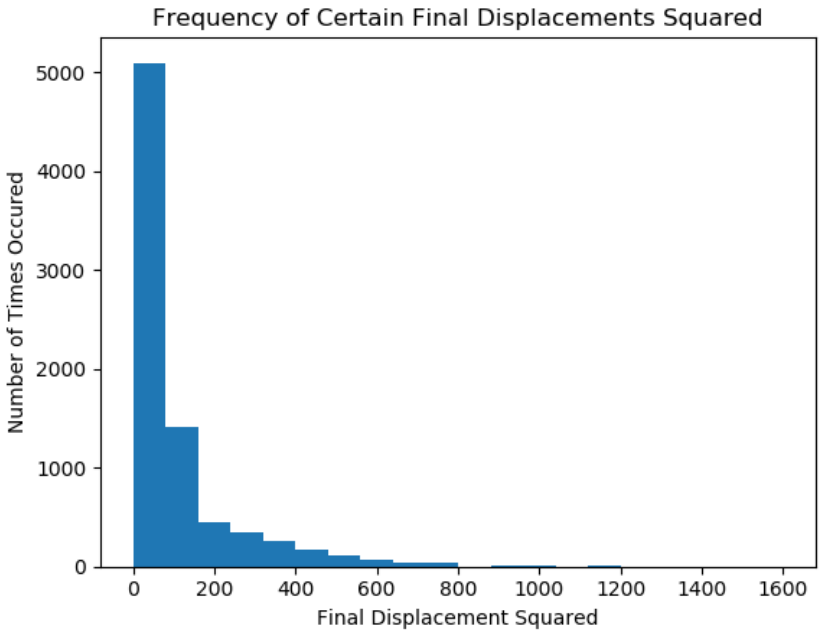
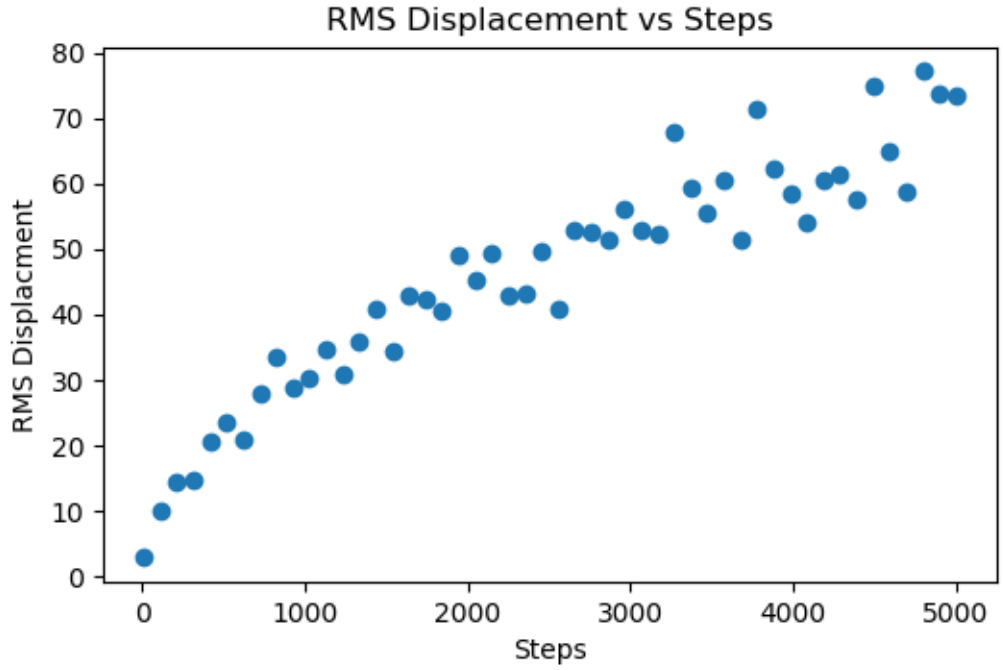
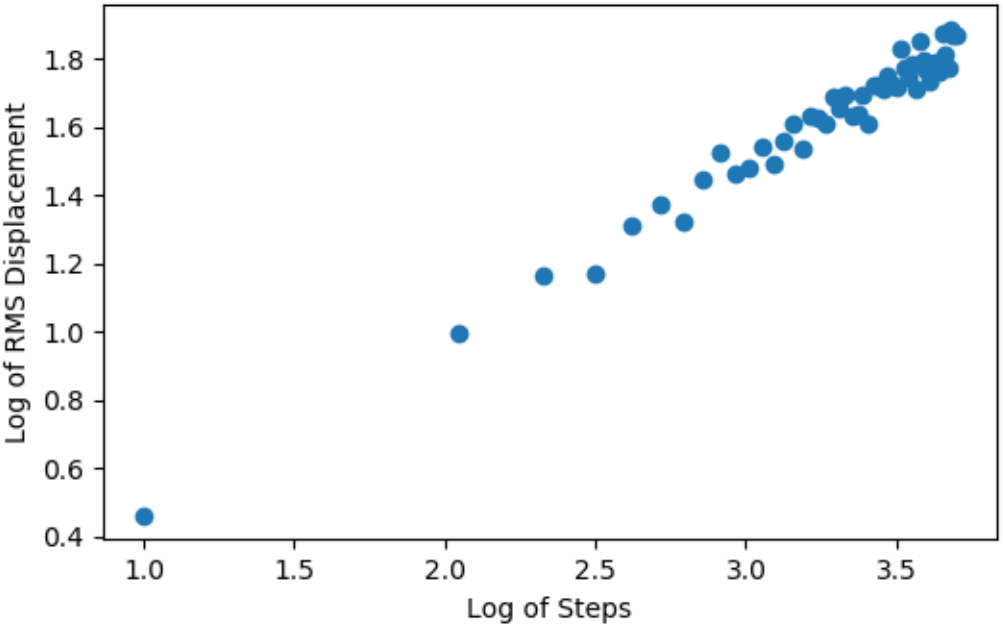


Figure 3: I want to see R1D1 on the one dimensional dance floor.




Frequency of Certain Final Displacements Squared
Number of Times Occured
5000
4000
3000
2000
1000
0
0
200
400
600
800
1000
1200
1400
1600
Final Displacement Squared


RMS Displacement vs Steps
RMS Displacment
Steps
0
10
20
30
40
50
60
70
80
0
1000
2000
3000
4000
5000


Log of RMS Displacement
Log of Steps
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1.8
1.0
1.5
2.0
2.5
3.0
3.5