[ ]:
```
import numpy as np
import matplotlib.pyplot as pyplot

#Important stable states
if(False):
    ziling,sid,avery = 1.0,1.0,1.0
    x = np.array([-1,0,1])
    y = np.array([-1,0,1])
    z = np.array([0,0,0])
    vx = np.array([1,0,-1])
    vy = np.array([-1,0,1])
    vz = np.array([0,0,0])

if(True):
    ziling,sid,avery = 1.0,0.5,2.0
    x = np.array([-1,0,0])
    y = np.array([-1,0,1])
    z = np.array([0,0,1])
    vx = np.array([0,0,0])
    vy = np.array([-1,0,1])
    vz = np.array([0,-1,0])
```

[ ]:
```
import numpy as np
import matplotlib.pyplot as pyplot

#Hi this is Zi, they locked me up to write code, so instead of explain␣
↪everything to you in the report
#I will explain them here in my code
#This block is where the set up for all the initial condition and the constants␣
↪happen

#Gravitational constant
g = 3
delta_t = 0.0001


#three masses
#in this case we are working with the three groupmembers in space
#imagine they are perfect point masses
#ziling,sid,avery = 1.0,1.0,2.0
m = np.array([ziling,sid,avery])

#initial positions and velociy

# index 0 -> m1
# index 1 -> m2
# index 2 -> m3
```

```
#so x = [x1,x2,x3]
#Avery will have the initial position of [x[2],y[2],z[2]]
#x = np.array([-1,0,1])
#y = np.array([-1,0,1])
#z = np.array([0,0,0])
#vx = np.array([1,0,-1])
#vy = np.array([-1,0,1])
#vz = np.array([0,0,0])

#Correction for linear momentum
lin_momentum = np.array([0,0,0])
for i in range(3):
    lin_momentum = lin_momentum + m[i]*np.array([vx[i],vy[i],vz[i]])

V = lin_momentum / (m[0] + m[1] + m[2])

for i in range(3):
    ones = 1/3 * np.array([1.0,1.0,1.0])
    vx = vx - V[0]*ones
    vy = vy - V[1]*ones
    vz = vz - V[2]*ones

#the dataset for the three objects
obj1,obj2,obj3 = [],[],[]
objs = [obj1,obj2,obj3]
for i in range(0,3):
    objs[i] = [m[i],x[i],y[i],z[i],vx[i],vy[i],vz[i]]




#Here is all the functions


def setInit(X,Y,Z,Vx,Vy,Vz, reset=True):
    """
    Sets the inititial values for the vectors
    @param X,Y,Z,Vx,Vy,Vz  the new initial values
    """
    global x,y,z,vx,vy,vz
    x,y,z,vx,vy,vz = np.array(X),np.array(Y),np.array(Z),np.array(Vx),np.
↪array(Vy),np.array(Vz)
    if(reset):
        resetVals()

def resetVals():
    """
```

```
    Resets the value in objs vector into the initial values
    """
    global objs
    global initial_k
    for i in range(0,3):
        objs[i] = [m[i],x[i],y[i],z[i],vx[i],vy[i],vz[i]]
    #objs = list(np.array(objs).copy())
    initial_k = constructK()

def r(m1,m2):
    """
    The distance between two bodies
    @param m1, m2 the two bodies
    @return the norm between the two distance vectors
    """
    return np.linalg.norm(np.array(objs[m1][1:4]) - np.array(objs[m2][1:4]))


def force(m):
    """
    Gravitational force caused by each body

    @param: m - The index of the object being calculated on
    @return the force on this object
    """
    s1 = (m+1)%3
    s2 = (m+2)%3

    f1 = g*objs[m][0]*objs[s1][0]/(r(m,s1)**3)
    f2 = g*objs[m][0]*objs[s2][0]/(r(m,s2)**3)

    r1 = np.
 ↪array([objs[s1][1]-objs[m][1],objs[s1][2]-objs[m][2],objs[s1][3]-objs[m][3]])
    r2 = np.
 ↪array([objs[s2][1]-objs[m][1],objs[s2][2]-objs[m][2],objs[s2][3]-objs[m][3]])

    return f1*r1 + f2*r2

def a(m):
    """
    Acceleration of each body
    @param: m - index of the object being calculated on
    @return: the acceleration of the object
    """
    return force(m)/objs[m][0]
```

```python
#refer back to report pg.6 for the matrix

def constructA():
    """
    just contruting A
    @return A - refer to chapter 5
    """

    c1 = []
    for i in range(0,3):
        #the force
        f = force(i)
        for j in range(0,3):
            c1.append(-f[j])
    for i in range(0,3):
        for j in range(4,len(objs[1])):
            c1.append(objs[i][0]*objs[i][j])

    c2,c3,c4 = [], [], []
    for i in range (0,9):
        c2.append(0)
        c3.append(0)
        c4.append(0)

    for i in range(0,3):
        for j in range (0,3):
            if(j==0):
                c2.append(objs[i][0])
            else:
                c2.append(0)
            if(j==1):
                c3.append(objs[i][0])
            else:
                c3.append(0)
            if(j==2):
                c4.append(objs[i][0])
            else:
                c4.append(0)

    c5,c6,c7 = [], [], []

    for i in range(0,3):
        for j in range (0,3):
            if(j==0):
                c5.append(objs[i][0])
            else:
                c5.append(0)
```

```
            if(j==1):
                c6.append(objs[i][0])
            else:
                c6.append(0)
            if(j==2):
                c7.append(objs[i][0])
            else:
                c7.append(0)

    for i in range (0,9):
        c5.append(0)
        c6.append(0)
        c7.append(0)



    c8 =␣
 ↪[0,m[0]*objs[0][6],-m[0]*objs[0][5],0,m[1]*objs[1][6],-m[1]*objs[1][5],0,m[2]*objs[2][6],-m[2
        ␣
 ↪0,-m[0]*objs[0][3],m[0]*objs[0][2],0,-m[1]*objs[1][3],m[1]*objs[1][2],0,-m[2]*objs[2][3],m[2]
    c9 =␣
 ↪[-m[0]*objs[0][6],0,m[0]*objs[0][4],-m[1]*objs[1][6],0,m[1]*objs[1][4],-m[2]*objs[2][6],0,m[2
        ␣
 ↪m[0]*objs[0][3],0,-m[0]*objs[0][1],m[1]*objs[1][3],0,-m[1]*objs[1][1],m[2]*objs[2][3],0,-m[2]
    c10 =␣
 ↪[m[0]*objs[0][5],-m[0]*objs[0][4],0,m[1]*objs[1][5],-m[1]*objs[1][4],0,m[2]*objs[2][5],-m[2]*
         ␣
 ↪-m[0]*objs[0][2],m[0]*objs[0][1],0,-m[1]*objs[1][2],m[1]*objs[1][1],0,-m[2]*objs[2][2],m[2]*o

    #my error matrix
    #this will be used to correct the states
    #its silly

    #the 10 rows are the 10 conserved quantity
    A = np.array([(c1),(c2),(c3),(c4),(c5),(c6),(c7),(c8),(c9),(c10)])

#now we are "normalizing" the matrix

    for i in range(10):
        magsqr = np.dot(A[i],A[i])
        if magsqr != 0:
            A[i] = A[i]/magsqr
    return A #now we have 'normalized' A



def constructK():
```

```
    """
    contruct the vector K, the correction for conserved quantities
    K = [energy, x momentum, y momentum, z momentum, center of mass x, center of␣
↪mass y, center of mass z, angular momentum x, angular momentum y, angular␣
↪momentum z]
    @return: The vector K (refer to chapter 5)
    """
    PE = 0
    for i in range(3):
        for j in range(3):
            if i != j:
                PE += -g*(m[i]*m[j])/np.linalg.norm(np.array(objs[i][1:4]) - np.
↪array(objs[j][1:4]))
    KE = 0
    for i in range(3):
        KE = (1/2)*m[i]*np.dot(np.array(objs[i][4:]),np.array(objs[i][4:]))
    E = KE + PE/4

    px,py,pz,comx,comy,comz= 0,0,0,0,0,0
    for i in range(3):
        px += m[i]*objs[i][4]
        py += m[i]*objs[i][5]
        pz += m[i]*objs[i][6]
        comx += m[i]*objs[i][1]
        comy += m[i]*objs[i][2]
        comz += m[i]*objs[i][3]

    omega =np.array([0,0,0])
    for i in range(3):
        omega = omega + (objs[i][0]*(np.cross(np.array(objs[i][1:4]),np.
↪array(objs[i][4:7]))))

    K = [E,px,py,pz,comx,comy,comz,omega[0],omega[1],omega[2]]

    return np.array(K)

#An important constant initial K
initial_K = constructK()
```

[ ]:
```
def update(delay = False):
    """
    The update function that updates the values inside objs vectors
    """

    #using Euler's method, we can update the values using the method below
    #a1=f(x1)
    #x2 = x1 + v1t
```

```
    #v2 = v1 + a1t
    global initial_K,delta_t
    accel = []
    for i in range(3):
        accel.append(a(i))
    #Updating the velocity of each object
    # m1 -> vx
    #update position of each object
    for i in range(3):
        objs[i][1] = objs[i][4]*delta_t + objs[i][1]
        objs[i][2] = objs[i][5]*delta_t + objs[i][2]
        objs[i][3] = objs[i][6]*delta_t + objs[i][3]
    #update veclocity of each object
    for i in range(3):
        objs[i][4] = accel[i][0]*delta_t + objs[i][4]
        objs[i][5] = accel[i][1]*delta_t + objs[i][5]
        objs[i][6] = accel[i][2]*delta_t + objs[i][6]

    if delay:
        for i in range(10):
            A = constructA()

            delta_K = (initial_K - constructK())
            delta_q = 0.01*np.matmul(np.matrix.transpose(A),delta_K)

            objs[0][1:4] += delta_q[:3]
            objs[1][1:4] += delta_q[3:6]
            objs[2][1:4] += delta_q[6:9]
            objs[0][4:7] += delta_q[9:12]
            objs[1][4:7] += delta_q[12:15]
            objs[2][4:7] += delta_q[15:18]

        return delta_q
```

[ ]:
```
#this is where the datas gets logged
log = []

resetVals()

temp_k = []
dqLog = []
time = []
t = 0
for i in range(100000):
    dq = update(True)
    objsLog = list(np.array(objs).copy())
    log.append(objsLog)
```

```
    time.append(t)
    t+=delta_t
    dqLog.append(dq)
    temp_k.append(constructK())
```

[ ]:
```
objects = list(zip(*log))

#setting up an 2x2 matrix of graph
fig, axs = pyplot.subplots(2,2)

for i in range(3):
    obj = objects[i]
    state = list(zip(*obj))
    x,y,z = state[1],state[2],state[3]
    axs[0,0].plot(x,y)
    axs[1,0].plot(x,z)
    axs[0,1].plot(z,y)

    axs[0,0].scatter(x[-1],y[-1])
    axs[1,0].scatter(x[-1],z[-1])
    axs[0,1].scatter(z[-1],y[-1])
axs[1,1].plot(time, list(zip(*temp_k))[0] - initial_K[0])


axs[0,0].set(xlabel = "x", ylabel = "y",title = "Y V.S X")
axs[1,0].set(xlabel = "x", ylabel = "z",title = "Z V.S X")
axs[0,1].set(xlabel = "z", ylabel = "y",title = "Y V.S Z")
axs[1,1].set(xlabel = "t", ylabel = "E",title = "Energy")
fig.tight_layout()
```

[ ]:
```
objects = list(zip(*log))

#set up the 3x3 matrix for linear momentum, average position and angular␣
↪momentum
fig, axs = pyplot.subplots(3,3)


for j in range(3):
    axs[0,j].plot(time, list(zip(*temp_k))[j+1] - initial_K[j+1])
    axs[1,j].plot(time, list(zip(*temp_k))[j+4] - initial_K[j+4])
    axs[2,j].plot(time, list(zip(*temp_k))[j+7] - initial_K[j+7])


axs[0,0].set(xlabel = "t", ylabel = "x",title = "Linear Momentum")
axs[0,1].set(xlabel = "t", ylabel = "y",title = "Linear Momentum")
axs[0,2].set(xlabel = "t", ylabel = "z",title = "Linear Momentum")
axs[1,0].set(xlabel = "t", ylabel = "x",title = "Average Position")
```

```
axs[1,1].set(xlabel = "t", ylabel = "y",title = "Average Position")
axs[1,2].set(xlabel = "t", ylabel = "z",title = "Average Position")
axs[2,0].set(xlabel = "t", ylabel = "x",title = "Angular Momentum")
axs[2,1].set(xlabel = "t", ylabel = "y",title = "Angular Momentum")
axs[2,2].set(xlabel = "t", ylabel = "z",title = "Angular Momentum")
fig.tight_layout()
```

```
[ ]: #The code for simulation is now done, the code below are for analyzing chaos



def r(ob, index1, index2):
    return np.sqrt((ob[index1][1]-ob[index2][1])**2 +␣
↪(ob[index1][2]-ob[index2][2])**2 + (ob[index1][3]-ob[index2][3])**2)

def relativeForce(ob, index1, index2):
    return ob[index1][0] * ob[index2][0] / ((r(ob, index1, index2))**3) \
            * np.array([ob[index2][1] - ob[index1][1], ob[index2][2] -␣
↪ob[index1][2], ob[index2][3] - ob[index1][3]])

def relativePotential(ob, index1, index2):
    return -ob[index1][0] * ob[index2][0] / r(ob, index1, index2)

def linearizBloc(ob, index1, index2):
    Fx, Fy, Fz= relativeForce(ob, index1, index2)
    U = relativePotential(ob, index1, index2)

    F = 3 * np.array([[Fx, Fy, Fz],\
                     [Fx, Fy, Fz],\
                     [Fx, Fy, Fz]])
    B = F + U * np.identity(3)
    B *= 1/r(ob, index1, index2)**2
    return B

def linearized(ob):
    Q = lambda i,j: linearizBloc(ob, i-1,j-1)
    A = np.block([[-Q(1,2)-Q(1,3),    Q(1,2)    ,    Q(1,3)    ],\
                  [    Q(2,1)    ,-Q(2,1)-Q(2,3),    Q(2,3)    ],\
                  [    Q(3,1)    ,    Q(3,2)    ,-Q(3,1)-Q(3,2)]])
    return A

def eigenvalues(ob):
    A = linearized(ob)
    eigenvalues = np.linalg.eigvals(A)
    return np.emath.sqrt(eigenvalues).real

def LyapunovExp(ob):
    A = linearized(ob)
```

```
    eigenvalues = np.linalg.eigvals(A)
    return max(np.emath.sqrt(eigenvalues).real)

'''---------------------------------------------------'''

def testEig(mean = 0, sigma = 1, dist = Gaussian, randomize = True):
    if(randomize):
        randomizeSettings(sigma, mean, dist)
    else:
        setInit(X = [1,0,0], Y = [0,2,0], Z = [0,0,3], Vx = [0,0,0], Vy =␣
 ↪[0,0,0], Vz = [0,0,0])

    A = linearized()
    print(A)
    print('\n-----------------------------------------------------\n')

    eigvals = eigenvalues()
    print(eigvals)

    print('\n-----------------------------------------------------\n')

    print(LyapunovExp())

def plotSpectra(batches, batchSize, mean = 0, sigma = 1, dist = Gaussian,␣
 ↪normalize = True):
    from functools import reduce

    N = batches * batchSize
    spectra = np.zeros(9)
    batch = np.zeros(9)

    for i in range(1,N):
        if(i % batchSize == 0):
            spectra += batch
            pyplot.plot(batch)
            batch = np.zeros(9)
            print(i)
        randomizeSettings(sigma, mean)
        eigvals = eigenvalues()
        if(normalize):
            total = sum(eigvals)
        else:
            total = 1
        if(total == 0):
            print(eigvals)
        else:
            batch += np.sort(eigvals / total)
```

```
    spectra /= N
    pyplot.show()
    pyplot.plot(spectra)

def plotMax(samples, smoothDens, mean = 0, sigma = 1, dist = Gaussian):

    vals = []

    for i in range(0,samples):
        randomizeSettings(sigma, mean, dist)
        eigvals = eigenvalues()
        vals.append(sum(eigvals) / 9)

    vals = np.sort(vals)
    #pyplot.plot(vals)
    #pyplot.show()

    density = smoothDens / (vals[smoothDens:-1]-vals[0:-smoothDens-1])
    density /= samples

    vals = vals[smoothDens//2:(-smoothDens)//2-1]

    return vals, density
```

[ ]:
```
#This is our reduced log
r_log = []
r_time = []

new_length = 10000

for i in range(0, len(log), len(log) // new_length):
    r_log.append(log[i])
    r_time.append(time[i])

print(len(r_log))
```

[ ]:
```
#list of eigenvalues per time
eigenvals = []
avg = []
mx = []
dev = []

for i in range(len(r_log)):
    eigen_values = eigenvalues(r_log[i])
    eigenvals.append(eigen_values)
```

```
    average = sum(eigen_values)/len(eigen_values)
    avg.append(average)

    maximum = 0
    for l in eigen_values:
        if(l>maximum):
            maximum = l
    mx.append(maximum)

    deviation = np.sqrt(sum([(l-average)**2 for l in eigen_values])/
↪len(eigen_values))
    dev.append(deviation)
```

[ ]:
```
# eigenvalues per size
eig = list(zip(*eigenvals))
for i in range(len(eig)):
    pyplot.plot(r_time, [np.exp(-eig[i][j]) for j in range(len(eig[i]))])
    #pyplot.show()
```

[ ]:
```
# eigenvalues per size
smooth = 500
eig = list(zip(*eigenvals))
for i in range(len(eig)):
    l = [sum([eig[i][k+j] for j in range(0,smooth)])/smooth for k in␣
↪range(len(eig[i])-smooth)]
    pyplot.plot(r_time[:-smooth], l)
    #pyplot.show()
```

[ ]:
```
# eigenvalues per size
eig = list(zip(*eigenvals))
pyplot.plot([sum(eig[i])/len(eig[i]) for i in range(len(eig))])
#pyplot.show()
```

[ ]:
```
# Average eigenvalue
pyplot.plot(r_time, avg)
    #pyplot.show()
```

[ ]:
```
# Average eigenvalue
pyplot.plot(r_time, mx)
    #pyplot.show()
```

[ ]:
```
# Average eigenvalue
pyplot.plot(r_time, dev)
    #pyplot.show()
```