```python
# -*- coding: utf-8 -*-
"""
Created on Sun Jun  3 11:19:56 2018

@author: Cemenenkoff

This code simulates N gravitationally interacting point masses in 3D. There are
five main sections:
    1. Main Switchboard
    2. Initial Conditions
    3. Main Function
    4. Data Generation
    5. Figures
"""
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from numpy import linalg as LA
plt.style.use('classic') #Use a serif font.
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('pdf', 'png')
plt.rcParams['savefig.dpi'] = 200
plt.rcParams['figure.autolayout'] = False
plt.rcParams['figure.figsize'] = 10, 6
plt.rcParams['axes.labelsize'] = 16
plt.rcParams['axes.titlesize'] = 20
plt.rcParams['font.size'] = 10
plt.rcParams['lines.linewidth'] = 2.0
plt.rcParams['lines.markersize'] = 6
plt.rcParams['legend.fontsize'] = 14
plt.rcParams['axes.facecolor'] = 'white'
plt.rcParams['savefig.facecolor']='white'
matplotlib.rcParams['xtick.direction'] = 'out'
matplotlib.rcParams['ytick.direction'] = 'out'
#This import is necessary for the isometric plot.
from mpl_toolkits.mplot3d import Axes3D
#Wrap tqdm() around the range() call in the method for-loops in orbit() to show
#the simulation's progress in the console with a growing horizontal bar.
from tqdm import tqdm
################################################################################
# 1. Main Switchboard ##########################################################
################################################################################
#Choose an integer number of bodies to explore. Make sure the chosen initial
#conditions data set is large enough for this choice of N.
N_ = 4

#Choose which set of initial conditions to explore (either 1, 2, or 3). IC1 is
#a set of conditions similar to our solar system while IC2 represents a more
#chaotic system. Note that max(N_)=10 for IC=1 and max(N_)=4 for IC=2. IC3 is a
#test case to see how the system operates under perfect symmetry (max(N_)=7).
IC = 2

#Choose a stepping method. Choices are:
#   'euler' (Euler)
#   'ec'    (Euler-Cromer)
#   'rk2'   (2nd Order Runge-Kutta)
#   'rk4'   (4th Order Runge-Kutta)
#   'vv'    (velocity Verlet)
#   'pv'    (position Verlet)
#   'vefrl' (velocity extended Forest-Ruth-like)
#   'pefrl' (position extended Forest-Ruth-like)
method_ = 'pv'
print(method_)

#Set time parameters for the simulation.
t0_ = 0.0 #start time in years
tf_ = 20.0 #final time in years
```

```python
dt_ = 2.0 #time step in hours

#Select a body index from 0, 1, ..., N-1. fig5-fig10 will be phase space plots
#for this selected body.
mf = 1

#Next, choose which figures to generate. The chosen figures will be shown in
#the console and also saved in the working directory.
gen_all = True #Set this switch to True to generate and save all plots.
gen_fig1 = True #3D plot of orbital trajectories
gen_fig2 = False #fig1 as viewed from the +x-axis
gen_fig3 = False #fig1 as viewed from the +y-axis
gen_fig4 = False #fig1 as viewed from the +z-axis
gen_fig5 = False #x-Component Momentum Phase Space for mf
gen_fig6 = False #y-Component Momentum Phase Space for mf
gen_fig7 = False #z-Component Momentum Phase Space for mf
gen_fig8 = False #x-Component Kinetic Energy Phase Space for mf
gen_fig9 = False #y-Component Kinetic Energy Phase Space for mf
gen_fig10 = False #z-Component Kinetic Energy Phase Space for mf
gen_fig11 = False #Total Kinetic Energy over Time for Each Body
gen_fig12 = True #Total Kinetic and Potential Energy of the System vs. Time
if gen_all == True:
    gen_fig1 = True
    gen_fig2 = True
    gen_fig3 = True
    gen_fig4 = True
    gen_fig5 = True
    gen_fig6 = True
    gen_fig7 = True
    gen_fig8 = True
    gen_fig9 = True
    gen_fig10 = True
    gen_fig11 = True
    gen_fig12 = True

##############################################################################
# 2. Initial Conditions ######################################################
##############################################################################
#Create a list of colors that is at least as long as the length of the largest
#mass list in the chosen IC set so each gravitationally interacting body has
#its own color.
c0 = '#ff0000' #red
c1 = '#8c8c94' #gray
c2 = '#ffd56f' #pale yellow
c3 = '#005e7b' #sea blue
c4 = '#a06534' #reddish brown
c5 = '#404436' #rifle green
c6 = '#7a26e7' #magenta
c7 = '#5CCE2A' #pale green
c8 = '#000000' #black
c9 = '#4542f4' #purple
#Put all of the colors into a global colors list.
c_ = [c0, c1, c2, c3, c4, c5, c6, c7, c8, c9]

#These initial conditions are similar to our own solar system, but the
#resulting orbital trajectories are not co-planar.
if IC == 1:
    #Define the masses for a number of bodies (in kg).
    m0 = 1.989e30 #mass of the sun
    m1 = 3.285e23 #mass of Mercury
    m2 = 4.867e24 #mass of Venus
    m3 = 5.972e24 #mass of Earth
    m4 = 6.417e23 #mass of Mars
    m5 = 1.898e27 #mass of Jupiter
    m6 = 5.683e26 #mass of Saturn
    m7 = 8.681e25 #mass of Uranus
    m8 = 1.024e26 #mass of Neptune
```

```python
        m9 = 1.309e22 #mass of Pluto

        #Combine the masses into a global masses list.
        m_ = [m0, m1, m2, m3, m4, m5, m6, m7, m8, m9]

        #What follows are two global initial conditions arrays. orbit() looks to
        #this data, but only imports the appropriate amount of rows for the given N.

        #For r0_, the first row of data represents m0's (x,y,z) initial position, the
        #second row represents m1's (x,y,z) initial position, etc.
        r0_ = np.array([[  1.0,  3.0,  2.0 ], #0
                        [  6.0, -5.0,  4.0 ], #1
                        [  7.0,  8.0, -7.0 ], #2
                        [  8.0,  6.0, -2.0 ], #3
                        [  8.8,  9.8, -6.8 ], #4
                        [  9.8,  3.8, -7.8 ], #5
                        [ -3.8,  1.8,  4.8 ], #6
                        [  7.8, -2.2,  1.8 ], #7
                        [  6.8, -4.1,  3.8 ], #8
                        [  5.8, -9.3,  5.8 ]])*1e11
        #For v0_, the first row of data represents m0's (x,y,z) initial velocity,
        #the second row represents m1's (x,y,z) initial velocity, etc.
        v0_ = np.array([[  0.0,  0.0,  0.0 ], #0
                        [  7.0,  0.5,  2.0 ], #1
                        [ -4.0, -0.5, -3.0 ], #2
                        [  7.0,  0.5,  2.0 ], #3
                        [  4.8,  1.3,  4.8 ], #4
                        [  1.8,  1.2, -5.8 ], #5
                        [  2.8, 11.3,  1.4 ], #6
                        [  3.8, 10.3,  2.4 ], #7
                        [  4.8,  9.3, -1.4 ], #8
                        [  5.8,  0.3, -2.4 ]])*1e3

    #This set of ICs is more like stars orbiting each other. The resulting orbital
    #trajectories are far less stable than those resulting from IC=1.
    if IC == 2:
        m0 = 1e30
        m1 = 2e30
        m2 = 3e30
        m3 = 2.5e30
        m_ = [m0, m1, m2, m3]
        r0_ = np.array([[  1.0,  3.0,  2.0],
                        [  6.0, -5.0,  4.0],
                        [  7.0,  8.0, -7.0],
                        [  8.0,  6.0, -2.0 ]])*1e11
        v0_ = np.array([[ -2.0,  0.5,  5.0],
                        [  7.0,  0.5,  2.0],
                        [ -4.0, -0.5, -3.0],
                        [  7.0,  0.5,  2.0 ]])*1e3

    #This set of ICs provides a test case where there is a central massive body,
    #and then 6 symmetrically arranged bodies surrounding it, all starting with
    #zero initial velocity.
    if IC == 3:
        m0 = 2e30
        m1 = 3.285e23
        m2 = 3.285e23
        m3 = 3.285e23
        m4 = 3.285e23
        m5 = 3.285e23
        m6 = 3.285e23
        m_ = [m0, m1, m2, m3, m4, m5, m6]
        r0_ = np.array([[  0.0,  0.0,  0.0], #0
                        [  1.0,  0.0,  0.0], #1
                        [  0.0,  1.0,  0.0], #2
                        [  0.0,  0.0,  1.0], #3
                        [ -1.0,  0.0,  0.0], #4
```

```
                     [  0.0, -1.0,  0.0], #5
                     [  0.0,  0.0, -1.0]])*1e11
    v0_ = np.array([[  0.0,  0.0,  0.0], #0
                     [  0.0,  0.0,  0.0], #1
                     [  0.0,  0.0,  0.0], #2
                     [  0.0,  0.0,  0.0], #3
                     [  0.0,  0.0,  0.0], #4
                     [  0.0,  0.0,  0.0], #5
                     [  0.0,  0.0,  0.0]])*1e3

################################################################################
# 3. Main Function #############################################################
################################################################################
"""
Purpose:
    orbit() calculates the orbital trajectories of N gravitationally
    interacting bodies given a set of mass and initial conditions data. Note
    both the mass list and initial conditions arrays must each contain data for
    at least N bodies, but may contain more. For example, orbit() could plot
    trajectories for the first 5 of 100 bodies in a large database.
Inputs:
    [0] N = the number of bodies to be considered in the calculation (integer)
    [1] t0 = the start time in years (number)
    [2] tf = the end time in years (number)
    [3] dt = the time step in hours (number)
    [4] m = list of masses of (at least) length N (list of numbers)
    [5] r0 = (at least) an Nx3 array of initial position data (2D numpy array)
    [6] v0 = (at least) an Nx3 array of initial velocity data (2D numpy array)
    [7] method = choice of stepping method (string)

Outputs:
    [0] r = position data for each body (3D numpy array)
    [1] v = velocity data for each body (3D numpy array)
    [2] p = momentum data for each body (3D numpy array)
    [3] KE = kinetic energy data for each body (3D numpy array)
    [4] T = total kinetic energy data for each body (3D numpy array)
    [5] Ts = total kinetic energy data for the system (2D numpy array)
    [6] Us = total potential energy data for the system (2D numpy array)
    [7] t = time data (1D numpy array)
"""
def orbit(N, t0, tf, dt, m, r0, v0, method):
    G = 6.67e-11 #universal gravitational constant in units of m^3/kg*s^2
    t0 = t0*365.26*24*3600 # Convert t0 from years to seconds.
    tf = tf*365.26*24*3600 # Convert tf from years to seconds.
    dt = dt*3600.0 #Convert dt from hours to seconds.
    steps = int(abs(tf-t0)/dt) #Define the total number of time steps.
    #Multiply an array of integers [(0, 1, ... , steps-1, steps)] by dt to get
    #an array of ascending time values.
    t = dt*np.array(range(steps))

    #If you print out either r or v below, you'll see several "layers" of Nx3
    #matrices. Which layer you are on represents which time step you are on.
    #Within each Nx3 matrix, the row denotes which body, while the columns 1-3
    #(indexed 0-2 in Python) represent x-, y-, z-positions respectively. In
    #essence, each number in r or v is associated with three indices:
    #step #, body #, and coordinate #.
    r = np.zeros([steps+1, N, 3])
    v = np.zeros([steps+1, N, 3])
    r[0] = r0[0:N] #Trim the initial conditions arrays to represent N bodies.
    v[0] = v0[0:N]

    """
    Purpose:
        accel() acts as a subroutine for orbit() by returning 3D acceleration
        vectors for a number of gravitationally interacting bodies given each
        of their positions.
    Input:
```

```python
        [0] r = 3D position vectors for all bodies at a certain time step
                (Nx3 numpy array)
    Output:
        [0] a = 3D acceleration vectors for each body (Nx3 numpy array)
    """
    def accel(r):
        a=np.zeros([N,3])
        #Each body's acceleration at each time step has to do with forces from
        #all other bodies. See: https://en.wikipedia.org/wiki/N-body_problem
        for i in range(N):
            #j is a list of indices of all bodies other than the ith body.
            j = list(range(i))+list(range(i+1,N))
            #Note each body's acceleration vector is a sum of N-1 terms, so for
            #each body, the terms are successively added to each other in a
            #running sum. Once all terms are added together, the ith body's
            #acceleration vector results.
            for k in range(N-1):
                a[i]=G*m[j[k]]/LA.norm(r[i]-r[j[k]])**3*(r[j[k]]-r[i])+a[i]
        return a

    #The simplest way to numerically integrate the accelerations into
    #velocities and then positions is with the Euler method. Note that this
    #method does not conserve energy.
    if method == 'euler':
        for i in tqdm(range(steps)):
            r[i+1] = r[i] + dt*v[i]
            v[i+1] = v[i] + dt*accel(r[i])

    #The Euler-Cromer method drives our next-simplest stepper.
    if method == 'ec':
        for i in tqdm(range(steps)):
            r[i+1] = r[i] + dt*v[i]
            v[i+1] = v[i] + dt*accel(r[i+1])

    #Getting slightly fancier, we employ the 2nd Order Runge-Kutta method.
    if method == 'rk2':
        for i in tqdm(range(steps)):
            v_iphalf = v[i] + accel(r[i])*(dt/2) # (i.e. v[i+0.5])
            r_iphalf = r[i] + v[i]*(dt/2)
            v[i+1] = v[i] + accel(r_iphalf)*dt
            r[i+1] = r[i] + v_iphalf*dt

    #Even fancier, here's the 4th Order Runge-Kutta method.
    if method == 'rk4':
        for i in tqdm(range(steps)):
            r1= r[i]
            v1 = v[i]
            a1 = accel(r1)
            r2 = r1+(dt/2)*v1
            v2 = v1+(dt/2)*a1
            a2 = accel(r2)
            r3 = r1+(dt/2)*v2
            v3 = v1+(dt/2)*a2
            a3 = accel(r3)
            r4 = r1+dt*v3
            v4 = v1+dt*a3
            a4 = accel(r4)
            r[i+1] = r[i]+(dt/6)*(v1 + 2*v2 + 2*v3 + v4)
            v[i+1] = v[i]+(dt/6)*(a1 + 2*a2 + 2*a3 + a4)

    #Here is a velocity Verlet implementation.
    #See: http://young.physics.ucsc.edu/115/leapfrog.pdf
    if method == 'vv':
        for i in tqdm(range(steps)):
            v_iphalf = v[i] + (dt/2)*accel(r[i])
            r[i+1] = r[i] + dt*v_iphalf
            v[i+1] = v_iphalf + (dt/2)*accel(r[i+1])

    #Next is a position Verlet implementation (found in the same pdf as 'vv').
    if method == 'pv':
        for i in tqdm(range(steps)):
            r_iphalf = r[i] + (dt/2)*v[i]
            v[i+1] = v[i] + dt*accel(r_iphalf)
            r[i+1] = r_iphalf + (dt/2)*v[i+1]

    #EFRL refers to an extended Forest-Ruth-like integration algorithm. Below
    #are three optimization parameters associated with EFRL routines.
    e = 0.1786178958448091e0
    l = -0.2123418310626054e0
    k = -0.6626458266981849e-1
    #First we do a velocity EFRL implementation (VEFRL).
    #See: https://arxiv.org/pdf/cond-mat/0110585.pdf
    if method == 'vefrl':
        for i in tqdm(range(steps)):
            v1 = v[i] + accel(r[i])*e*dt
            r1 = r[i] + v1*(1-2*l)*(dt/2)
            v2 = v1 + accel(r1)*k*dt
            r2 = r1 + v2*l*dt
            v3 = v2 + accel(r2)*(1-2*(k+e))*dt
            r3 = r2 + v3*l*dt
            v4 = v3 + accel(r3)*k*dt
            r[i+1] = r3 + v4*(1-2*l)*(dt/2)
            v[i+1] = v4 + accel(r[i+1])*e*dt

    #Next is a position EFRL (PEFRL) (found in the same pdf as 'vefrl').
    if method == 'pefrl':
        for i in tqdm(range(steps)):
            r1 = r[i] + v[i]*e*dt
            v1 = v[i] + accel(r1)*(1-2*l)*(dt/2)
            r2 = r1 + v1*k*dt
            v2 = v1 + accel(r2)*l*dt
            r3 = r2 + v2*(1-2*(k+e))*dt
            v3 = v2 + accel(r3)*l*dt
            r4 = r3 + v3*k*dt
            v[i+1] = v3 + accel(r4)*(1-2*l)*(dt/2)
            r[i+1] = r4 + v[i+1]*e*dt

    """
    Purpose:
        PE() acts as a subroutine for orbit() by returning the total
        gravitational potential energy for N bodies given their positions.
    Input:
        [0] r = 3D position vectors for all bodies at a certain time step
                (Nx3 numpy array)
    Output:
        [0] Us = total gravitational potential energy of the system (float)
    """
    def PE(r):
        Us = 0
        for j in range(N):
            i=0
            while i!=j:
                Us = -G*m[i]*m[j]/LA.norm(r[i]-r[j])+Us
                i+=1
        return Us

    #Derive the system's total potential energy data from the position data.
    Us = np.zeros((steps,1))
    for i in range(steps):
        Us[i] = PE(r[i])

    #Lastly, derive kinetic energy and momentum data from the velocity data.
    Ts = np.zeros((steps,1)) #total kinetic energy of the system
    T = np.zeros((steps,N,1)) #total kinetic energy data for each body
    KE = np.zeros((steps,N,3)) #3D kinetic energy data for each body
    p = np.zeros((steps,N,3)) #3D momentum data for each body
    v2 = v**2 #Square all velocities for use in the energy calculation.
    for i in range(steps):
        for j in range(N):
            KE[i,j,:] = (m[j]/2)*v2[i,j,:]
            T[i,j,0] = sum(KE[i,j,:])
            p[i,j,:] = m[j]*v[i,j,:]
        Ts[i] = sum(T[i])

    return r, v, p, KE, T, Ts, Us, t

################################################################################
# 4. Data Generation ###########################################################
################################################################################
r, v, p , KE, T, Ts, Us, t = orbit(N_, t0_, tf_, dt_, m_, r0_, v0_, method_)

################################################################################
# 5. Figures ###################################################################
################################################################################
mf_lab = '$m_'+str(mf)+'$' #Create a LaTeX-wrapped label for mf.

#Generate an ascending list of integers from 2 to N and then change the
#elements into strings for use in figure titles.
words = list(range(2,N_+1))
words = [str(x) for x in words]
#Wrap each string in LaTeX so it has a serif font on the plot.
for i in range(len(words)):
    words[i] = '$\mathrm{'+words[i]+'\ }$'
Nstr = words[N_-2] #Note the index shift because words[0]='2'.

labs = [None]*N_ #Create a list to store labels for the masses (m0, m1, etc.).
#Wrap each label with LaTeX math mode so it prints with a serif font.
for i in range(N_):
    labs[i] = r'$m_'+str(i)+'$'

a_ = 0.7 #Set a global transparency value so we can see where orbits overlap.
#Set low and high bounds for arrays used in the phase space figures to ensure
#the system is in a "settled down" dynamic equilibrium.
lo = int(0.35*(len(t)))
hi = int(0.95*(len(t)))
#-------------------------------------------------------------------------------
if gen_fig1 == True:
    fig1 = plt.figure(1, facecolor='white') #3D plot of orbital trajectories
    ax1 = fig1.add_subplot(1,1,1, projection='3d')
    plt.title(r'%s'%Nstr+r'$\mathrm{Orbiting\ Bodies}$', y=1.05)
    ax1.set_xlabel(r'$\mathrm{x-Position}\ \mathrm{(m)}$', labelpad=10)
    ax1.set_ylabel(r'$\mathrm{y-Position}\ \mathrm{(m)}$', labelpad=10)
    ax1.set_zlabel(r'$\mathrm{z-Position}\ \mathrm{(m)}$', labelpad=10)
    for i in range(N_): #For all times, plot mi's (x,y,z) data.
        ax1.plot(r[:, i, 0], r[:, i, 1], r[:, i, 2], color=c_[i],label=labs[i],
              alpha=a_)
    ax1.axis('equal')
    plt.legend(loc='upper left')
    plt.savefig('fig1', bbox_inches='tight')
#-------------------------------------------------------------------------------
if gen_fig2 == True:
    fig2 = plt.figure(2, facecolor='white') #fig1 as viewed from the +x-axis
    ax2 = fig2.add_subplot(111)
    plt.title(r'%s'%Nstr+r'$\mathrm{Orbiting\ Bodies\ }$'+'\n'
              +r'$\mathrm{as\ Viewed \ From\ the\ Positive\ x-Axis}$', y=1.05)
    ax2.set_xlabel(r'$\mathrm{y-Position}\ \mathrm{(m)}$')
    ax2.set_ylabel(r'$\mathrm{z-Position}\ \mathrm{(m)}$')
    for i in range(N_): #For all times, plot mi's (y,z) data.
        ax2.plot(r[:, i, 1], r[:, i, 2], color=c_[i], label=labs[i], alpha=a_)
    ax2.axis('equal')
    ax2.legend(loc='lower right')
    plt.savefig('fig2', bbox_inches='tight')
#-------------------------------------------------------------------------------
if gen_fig3 == True:
    fig3 = plt.figure(3, facecolor='white') #fig1 as viewed from the +y-axis
    ax3 = fig3.add_subplot(111)
    plt.title(r'%s'%Nstr+r'$\mathrm{Orbiting\ Bodies\ }$'+'\n'
              +r'$\mathrm{as\ Viewed \ From\ the\ Positive\ y-Axis}$', y=1.05)
    ax3.set_xlabel(r'$\mathrm{x-Position}\ \mathrm{(m)}$')
    ax3.set_ylabel(r'$\mathrm{z-Position}\ \mathrm{(m)}$')
    for i in range(N_): #For all times, plot mi's (x,z) data.
        ax3.plot(r[:, i, 0], r[:, i, 2], color=c_[i], label=labs[i], alpha=a_)
    ax3.axis('equal')
    ax3.legend(loc='lower right')
    plt.savefig('fig3', bbox_inches='tight')
#-------------------------------------------------------------------------------
if gen_fig4 == True:
    fig4 = plt.figure(4, facecolor='white') #fig1 as viewed from the +z-axis
    ax4 = fig4.add_subplot(111)
    plt.title(r'%s'%Nstr+r'$\mathrm{Orbiting\ Bodies\ }$'+'\n'
              +r'$\mathrm{as\ Viewed \ From\ the\ Positive\ z-Axis}$', y=1.05)
    ax4.set_xlabel(r'$\mathrm{x-Position}\ \mathrm{(m)}$')
    ax4.set_ylabel(r'$\mathrm{y-Position}\ \mathrm{(m)}$')
    for i in range(N_): #For all times, plot mi's (x,y) data.
        ax4.plot(r[:, i, 0], r[:, i, 1], color=c_[i], label=labs[i], alpha=a_)
    ax4.axis('equal')
    ax4.legend(loc='lower right')
    plt.savefig('fig4', bbox_inches='tight')

#-------------------------------------------------------------------------------
if gen_fig5 == True:
    #x-Component Momentum Phase Space for mf
    fig5 = plt.figure(5, facecolor='white')
    ax5 = fig5.add_subplot(111)
    plt.title(r'%s'%Nstr+r'$\mathrm{Orbiting\ Bodies\ }$'+'\n'
              +r'$\mathrm{x-Component\ Momentum\ Phase\ Space\ for\ }$'
              +r'%s'%mf_lab, y=1.05)
    ax5.set_xlabel(r'$\mathrm{x-Position}\ \mathrm{(m)}$')
    ax5.set_ylabel(r'$\mathrm{x-Momentum}\ \mathrm{(kg\cdot\frac{m}{s})}$')
    ax5.plot(r[lo:hi, mf, 0], p[lo:hi, mf, 0], color=c_[mf], label=labs[mf],
             alpha=a_)
    plt.savefig('fig5', bbox_inches='tight')
#-------------------------------------------------------------------------------
if gen_fig6 == True:
    #y-Component Momentum Phase Space for mf
    fig6 = plt.figure(6, facecolor='white')
    ax6 = fig6.add_subplot(111)
    plt.title(r'%s'%Nstr+r'$\mathrm{Orbiting\ Bodies\ }$'+'\n'
              +r'$\mathrm{y-Component\ Momentum\ Phase\ Space\ for\ }$'
              +r'%s'%mf_lab, y=1.05)
    ax6.set_xlabel(r'$\mathrm{y-Position}\ \mathrm{(m)}$')
    ax6.set_ylabel(r'$\mathrm{y-Momentum}\ \mathrm{(kg\cdot\frac{m}{s})}$')
    ax6.plot(r[lo:hi, mf, 1], p[lo:hi, mf, 1], color=c_[mf], label=labs[mf],
             alpha=a_)
    plt.savefig('fig6', bbox_inches='tight')
#-------------------------------------------------------------------------------
if gen_fig7 == True:
    #z-Component Momentum Phase Space for mf
    fig7 = plt.figure(7, facecolor='white')
    ax7 = fig7.add_subplot(111)
    plt.title(r'%s'%Nstr+r'$\mathrm{Orbiting\ Bodies\ }$'+'\n'
              +r'$\mathrm{z-Component\ Momentum\ Phase\ Space\ for\ }$'
              +r'%s'%mf_lab, y=1.05)
    ax7.set_xlabel(r'$\mathrm{z-Position}\ \mathrm{(m)}$')
    ax7.set_ylabel(r'$\mathrm{z-Momentum}\ \mathrm{(kg\cdot\frac{m}{s})}$')
    ax7.plot(r[lo:hi, mf, 2], p[lo:hi, mf, 2], color=c_[mf], label=labs[mf],
             alpha=a_)
    plt.savefig('fig7', bbox_inches='tight')
```

```python

#----------------------------------------------------------------------------
if gen_fig8 == True:
    #x-Component Kinetic Energy Phase Space for mf
    fig8 = plt.figure(8, facecolor='white')
    ax8 = fig8.add_subplot(111)
    plt.title(r'%s'%Nstr+r'$\mathrm{Orbiting\ Bodies}$'+'\n'
              +r'$\mathrm{x-Component\ Kinetic\ Energy\ Phase\ Space\ for\ }$'
              +r'%s'%mf_lab, y=1.05)
    ax8.set_xlabel(r'$\mathrm{x-Position}\ \mathrm{(m)}$')
    ax8.set_ylabel(r'$\mathrm{x-Kinetic\ Energy}\ \mathrm{(J)}$')
    ax8.plot(r[lo:hi, mf, 0], KE[lo:hi, mf, 0], color=c_[mf], label=labs[mf],
             alpha=a_)
    plt.savefig('fig8', bbox_inches='tight')
#----------------------------------------------------------------------------
if gen_fig9 == True:
    #y-Component Kinetic Energy Phase Space for mf
    fig9 = plt.figure(9, facecolor='white')
    ax9 = fig9.add_subplot(111)
    plt.title(r'%s'%Nstr+r'$\mathrm{Orbiting\ Bodies}$'+'\n'
              +r'$\mathrm{y-Component\ Kinetic\ Energy\ Phase\ Space\ for\ }$'
              +r'%s'%mf_lab, y=1.05)
    ax9.set_xlabel(r'$\mathrm{y-Position}\ \mathrm{(m)}$')
    ax9.set_ylabel(r'$\mathrm{y-Kinetic\ Energy}\ \mathrm{(J)}$')
    ax9.plot(r[lo:hi, mf, 1], KE[lo:hi, mf, 1], color=c_[mf], label=labs[mf],
             alpha=a_)
    plt.savefig('fig9', bbox_inches='tight')
#----------------------------------------------------------------------------
if gen_fig10 == True:
    #z-Component Kinetic Energy Phase Space for mf
    fig10 = plt.figure(10, facecolor='white')
    ax10 = fig10.add_subplot(111)
    plt.title(r'%s'%Nstr+r'$\mathrm{Orbiting\ Bodies}$'+'\n'
              +r'$\mathrm{z-Component\ Kinetic\ Energy\ Phase\ Space\ for\ }$'
              +r'%s'%mf_lab, y=1.05)
    ax10.set_xlabel(r'$\mathrm{z-Position}\ \mathrm{(m)}$')
    ax10.set_ylabel(r'$\mathrm{z-Kinetic\ Energy}\ \mathrm{(J)}$')
    ax10.plot(r[lo:hi, mf, 2], KE[lo:hi, mf, 2], color=c_[mf], label=labs[mf],
              alpha=a_)
    plt.savefig('fig10', bbox_inches='tight')

#----------------------------------------------------------------------------
if gen_fig11 == True:
    #Total Kinetic Energy over Time for Each Body
    fig11 = plt.figure(11, facecolor='white')
    ax11 = fig11.add_subplot(111)
    plt.title(r'%s'%Nstr+r'$\mathrm{Orbiting\ Bodies\ }$'+'\n'
              +r'$\mathrm{Kinetic\ Energy\ vs.\ Time}$', y=1.05)
    ax11.set_xlabel(r'$\mathrm{Time}\ \mathrm{(s)}$')
    ax11.set_ylabel(r'$\mathrm{Kinetic\ Energy}\ \mathrm{(J)}$')
    for i in range(N_):
        ax11.plot(t, T[:, i, :], color=c_[i], label=labs[i], alpha=a_)
    ax11.legend(loc='lower right')
    plt.savefig('fig11', bbox_inches='tight')
#----------------------------------------------------------------------------
if gen_fig12 == True:
    #Total Kinetic and Potential Energy of the System vs. Time
    fig12 = plt.figure(12, facecolor='white')
    ax12 = fig12.add_subplot(111)
    plt.title(r'%s'%Nstr+r'$\mathrm{Orbiting\ Bodies\ }$'+'\n'
              +r'$\mathrm{Total\ Kinetic\ and\ Potential\ }$'
              +r'$\mathrm{Energies \ vs.\ Time}$', y=1.05)
    ax12.set_xlabel(r'$\mathrm{Time}\ \mathrm{(s)}$')
    ax12.set_ylabel(r'$\mathrm{Energy}\ \mathrm{(J)}$')
    ax12.plot(t, Ts, color='black', label = r'$T_{\mathrm{total}}$', alpha=a_)
    ax12.plot(t, Us, color='red', label = r'$U_{\mathrm{total}}$', alpha=a_)
    ax12.legend(loc='lower right')
    plt.savefig('fig12', bbox_inches='tight')
```