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

@author: Cemenenkoff
"""
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
################################################################################
N_ = 5 #Choose a number of bodies to explore between 2 and 5.
#Choose a stepping method. Choices are:
#   'euler' (Euler)
#   'rk2'   (2nd Order Runge-Kutta)
#   'ec'    (Euler-Cromer)
#   'vv'    (velocity Verlet)
#   'pv'    (position Verlet)
#   'vefrl' (velocity extended Forest-Ruth-like)
#   'pefrl' (position extended Forest-Ruth-like)
method_ = 'pefrl'

################################################################################
G = 6.67e-11 #universal gravitational constant in units of m^3/kg*s^2
end_time = 100*365.26*24*3600 # Define the end time in seconds (e.g. 100 years).
dt = 2.0*3600 #Define the length of each time step in seconds (e.g. 2 hours).
steps = int(end_time/dt) #Define the total number of time steps.
#Fill out a list of times by successively adding increments of dt.
t = dt*np.array(range(steps + 1))

#Define the masses for a number of bodies (in kg).
m0 = 2.0e30 #mass of the sun
m1 = 3.285e23 #mass of Mercury
m2 = 4.8e24 #mass of Venus
m3 = 6.0e24 #mass of Earth
m4 = 2.4e24 #mass of Mars

m = [m0, m1, m2, m3, m4] #Combine the masses into an ordered list.
#Set colors for the bodies.
c0 = '#ff0000' #red
c1 = '#8c8c94' #gray
c2 = '#ffd56f' #pale yellow
c3 = '#005e7b' #sea blue
c4 = '#a06534' #reddish brown
c = [c0, c1, c2, c3, c4] #Put colors into a similarly ordered list.
################################################################################
#Orbit takes in a list of masses (floats), the number of bodies N (integer
#between 2 and 5), and the desired stepping method (string). This function lays
#a foundation for a general N-body function, but such a function's development
```

```
#will be saved for later if at all. Five bodies should be enough to get a
#general feel for how the problem would extend to N bodies.
def orbit(m, N, method):
    #If you print out either r or v below, you'll see several "layers" of 3x3
    #matrices. Which layer you are on represents which time step you are on.
    #Within each 3x3 matrix, the row denotes which body, while the columns 1-3
    #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])
    if N == 2:
        #Next, we input initial positions. Note the first bracketed triplet of
        #data represents the x-, y-, and z-position of the first body (m0).
        r[0] = np.array([[1.0, 3.0, 2.0],
                         [6.0, -5.0, 4.0]])*1e11
        #Input initial velocities. Note the sun has zero velocity here.
        v[0] = np.array([[0.0, 0.0, 0.0],
                             [7.0, 0.5, 2.0]])*1e3
    if N == 3:
        r[0] = np.array([[1.0, 3.0, 2.0],
                             [6.0, -5.0, 4.0],
                             [7.0, 8.0, -7.0]])*1e11
        v[0] = np.array([[0.0, 0.0, 0.0],
                             [7.0, 0.5, 2.0],
                             [-4.0, -0.5, -3.0]])*1e3
    if N == 4:
        r[0] = np.array([[1.0, 3.0, 2.0],
                         [6.0, -5.0, 4.0],
                         [7.0, 8.0, -7.0],
                         [8.0, 9.0, -6.0]])*1e11
        v[0] = np.array([[0.0, 0.0, 0.0],
                             [7.0, 0.5, 2.0],
                             [-4.0, -0.5, -3.0],
                             [7.0, 0.5, 2.0]])*1e3
    if N == 5:
        r[0] = np.array([[1.0, 3.0, 2.0],
                         [6.0, -5.0, 4.0],
                         [7.0, 8.0, -7.0],
                         [8.0, 9.0, -6.0],
                         [8.8, 9.8, -6.8]])*1e11
        v[0] = np.array([[0.0, 0.0, 0.0],
                             [7.0, 0.5, 2.0],
                             [-4.0, -0.5, -3.0],
                             [7.0, 0.5, 2.0],
                             [7.8, 1.3, 2.8]])*1e3

    #Each body's acceleration at each time step has to do with the force from
    #all other bodies and vice versa. It can be seen that this formula is
    #generalizable, but doing so is beyond the scope of this exploration.
    #See: https://en.wikipedia.org/wiki/N-body_problem
    def accel(r):
        a = np.zeros([N,3])
        if N == 2:
            a[0] = G*(m[1]/LA.norm(r[0]-r[1])**3*(r[1]-r[0]))
            a[1] = G*(m[0]/LA.norm(r[1]-r[0])**3*(r[0]-r[1]))
        if N == 3:
            a[0] = G*(m[1]/LA.norm(r[0]-r[1])**3*(r[1]-r[0])
                    + m[2]/LA.norm(r[0]-r[2])**3*(r[2]-r[0]))

            a[1] = G*(m[0]/LA.norm(r[1]-r[0])**3*(r[0]-r[1])
                    + m[2]/LA.norm(r[1]-r[2])**3*(r[2]-r[1]))

            a[2] = G*(m[0]/LA.norm(r[2]-r[0])**3*(r[0]-r[2])
                    + m[1]/LA.norm(r[2]-r[1])**3*(r[1]-r[2]))
        if N == 4:
            a[0] = G*(m[1]/LA.norm(r[0]-r[1])**3*(r[1]-r[0])
                    + m[2]/LA.norm(r[0]-r[2])**3*(r[2]-r[0])
```

```
                    + m[3]/LA.norm(r[0]-r[3])**3*(r[3]-r[0]))

            a[1] = G*(m[0]/LA.norm(r[1]-r[0])**3*(r[0]-r[1])
                    + m[2]/LA.norm(r[1]-r[2])**3*(r[2]-r[1])
                    + m[3]/LA.norm(r[1]-r[3])**3*(r[3]-r[1]))

            a[2] = G*(m[0]/LA.norm(r[2]-r[0])**3*(r[0]-r[2])
                    + m[1]/LA.norm(r[2]-r[1])**3*(r[1]-r[2])
                    + m[3]/LA.norm(r[2]-r[3])**3*(r[3]-r[2]))

            a[3] = G*(m[0]/LA.norm(r[3]-r[0])**3*(r[0]-r[3])
                    + m[1]/LA.norm(r[3]-r[1])**3*(r[1]-r[3])
                    + m[2]/LA.norm(r[3]-r[2])**3*(r[2]-r[3]))
        if N == 5:
            a[0] = G*(m[1]/LA.norm(r[0]-r[1])**3*(r[1]-r[0])
                    + m[2]/LA.norm(r[0]-r[2])**3*(r[2]-r[0])
                    + m[3]/LA.norm(r[0]-r[3])**3*(r[3]-r[0])
                    + m[4]/LA.norm(r[0]-r[4])**3*(r[4]-r[0]))

            a[1] = G*(m[0]/LA.norm(r[1]-r[0])**3*(r[0]-r[1])
                    + m[2]/LA.norm(r[1]-r[2])**3*(r[2]-r[1])
                    + m[3]/LA.norm(r[1]-r[3])**3*(r[3]-r[1])
                    + m[4]/LA.norm(r[1]-r[4])**3*(r[4]-r[1]))

            a[2] = G*(m[0]/LA.norm(r[2]-r[0])**3*(r[0]-r[2])
                    + m[1]/LA.norm(r[2]-r[1])**3*(r[1]-r[2])
                    + m[3]/LA.norm(r[2]-r[3])**3*(r[3]-r[2])
                    + m[4]/LA.norm(r[2]-r[4])**3*(r[4]-r[2]))

            a[3] = G*(m[0]/LA.norm(r[3]-r[0])**3*(r[0]-r[3])
                    + m[1]/LA.norm(r[3]-r[1])**3*(r[1]-r[3])
                    + m[2]/LA.norm(r[3]-r[2])**3*(r[2]-r[3])
                    + m[4]/LA.norm(r[3]-r[4])**3*(r[4]-r[3]))

            a[4] = G*(m[0]/LA.norm(r[4]-r[0])**3*(r[0]-r[4])
                    + m[1]/LA.norm(r[4]-r[1])**3*(r[1]-r[4])
                    + m[2]/LA.norm(r[4]-r[2])**3*(r[2]-r[4])
                    + m[3]/LA.norm(r[4]-r[3])**3*(r[3]-r[4]))
        return a

    #The simplest implementation is the Euler method. Note that this method
    #does not conserve energy.
    if method == 'euler':
        for i in range(steps):
            r[i+1] = r[i] + dt*v[i]
            v[i+1] = v[i] + dt*accel(r[i])


    #The next-simplest method is an Euler-Cromer implementation.
    if method == 'ec':
        for i in 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 range(steps):
            v_iphalf = v[i] + accel(r[i])*(dt/2)
            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

    #Here is a velocity Verlet implementation.
    #See: http://young.physics.ucsc.edu/115/leapfrog.pdf
    if method == 'vv':
        for i in 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 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 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':
            e = 0.1786178958448091e0
            l = -0.2123418310626054e0
            k = -0.6626458266981849e-1
            for i in 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

        return r, v

    ##############################################################################
    r, v = orbit(m, N_, method_)
    if N_ == 2:
        #r, v = two(m, method_)
        Nstr = '$\mathrm{Two\ }$'
    elif N_ == 3:
        #r, v = three(m, method_)
        Nstr = '$\mathrm{Three\ }$'
    elif N_ == 4:
        #r, v = four(m, method_)
        Nstr = '$\mathrm{Four\ }$'
    elif N_ == 5:
        #r, v = five(m, method_)
        Nstr = '$\mathrm{Five\ }$'
    ##############################################################################
    fig1 = plt.figure(1, facecolor='white')
    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)
```

```python
ax1.set_zlabel(r'$\mathrm{z-position}\ \mathrm{(m)}$', labelpad=10)

a_ = 0.7 #Set a transparency value so we can see where orbits overlap.
#For all times, plot (x,y,z) tuples for m0.
ax1.plot(r[:, 0, 0], r[:, 0, 1], r[:, 0, 2], color=c[0], label=r'$m_0$',
         alpha=a_)
#For all times, plot (x,y,z) tuples for m1, etc.
ax1.plot(r[:, 1, 0], r[:, 1, 1], r[:, 1, 2], color=c[1], label=r'$m_1$',
         alpha=a_)
if N_ > 2:
    ax1.plot(r[:, 2, 0], r[:, 2, 1], r[:, 2, 2], color=c[2], label=r'$m_2$',
             alpha=a_)
if N_ > 3:
    ax1.plot(r[:, 3, 0], r[:, 3, 1], r[:, 3, 2], color=c[3], label=r'$m_3$',
             alpha=a_)
if N_ > 4:
    ax1.plot(r[:, 4, 0], r[:, 4, 1], r[:, 4, 2], color=c[4], label=r'$m_4$',
             alpha=a_)
ax1.axis('equal')
plt.legend(loc='upper left')
###############################################################################
fig2 = plt.figure(2, facecolor='white')
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 all times, plot (x,z) tuples for m0.
ax2.plot(r[:, 0, 1], r[:, 0, 2], color=c[0], label=r'$m_0$', alpha=a_)
#For all times, plot (x,z) tuples for m1, etc.
ax2.plot(r[:, 1, 1], r[:, 1, 2], color=c[1], label=r'$m_1$', alpha=a_)
if N_ > 2:
    ax2.plot(r[:, 2, 1], r[:, 2, 2], color=c[2], label=r'$m_2$', alpha=a_)
if N_ > 3:
    ax2.plot(r[:, 3, 1], r[:, 3, 2], color=c[3], label=r'$m_3$', alpha=a_)
if N_ > 4:
    ax2.plot(r[:, 4, 1], r[:, 4, 2], color=c[4], label=r'$m_4$', alpha=a_)
ax2.axis('equal')
ax2.legend(loc='lower right')
###############################################################################
fig3 = plt.figure(3, facecolor='white')
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 all times, plot (x,z) tuples for m0.
ax3.plot(r[:, 0, 0], r[:, 0, 2], color=c[0], label=r'$m_0$', alpha=a_)
#For all times, plot (x,z) tuples for m1, etc.
ax3.plot(r[:, 1, 0], r[:, 1, 2], color=c[1], label=r'$m_1$', alpha=a_)
if N_ > 2:
    ax3.plot(r[:, 2, 0], r[:, 2, 2], color=c[2], label=r'$m_2$', alpha=a_)
if N_ > 3:
    ax3.plot(r[:, 3, 0], r[:, 3, 2], color=c[3], label=r'$m_3$', alpha=a_)
if N_ > 4:
    ax3.plot(r[:, 4, 0], r[:, 4, 2], color=c[4], label=r'$m_4$', alpha=a_)
ax3.axis('equal')
ax3.legend(loc='lower right')
###############################################################################
fig4 = plt.figure(4, facecolor='white')
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 all times, plot (x,z) tuples for m0.
ax4.plot(r[:, 0, 0], r[:, 0, 1], color=c[0], label=r'$m_0$', alpha=a_)
#For all times, plot (x,z) tuples for m1, etc.
ax4.plot(r[:, 1, 0], r[:, 1, 1], color=c[1], label=r'$m_1$', alpha=a_)
if N_ > 2:
    ax4.plot(r[:, 2, 0], r[:, 2, 1], color=c[2], label=r'$m_2$', alpha=a_)
if N_ > 3:
    ax4.plot(r[:, 3, 0], r[:, 3, 1], color=c[3], label=r'$m_3$', alpha=a_)
if N_ > 4:
    ax4.plot(r[:, 4, 0], r[:, 4, 1], color=c[4], label=r'$m_4$', alpha=a_)
ax4.axis('equal')
ax4.legend(loc='lower right')
```