

```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Sun Jun  3 11:19:56 2018
4
5  @author: Cemenenkoff
6  """
7  import matplotlib
8  import matplotlib.pyplot as plt
9  import numpy as np
10 from numpy import linalg as LA
11 plt.style.use('classic') #Use a serif font.
12 from IPython.display import set_matplotlib_formats
13 set_matplotlib_formats('pdf', 'png')
14 plt.rcParams['savefig.dpi'] = 200
15 plt.rcParams['figure.autolayout'] = False
16 plt.rcParams['figure.figsize'] = 10, 6
17 plt.rcParams['axes.labelsize'] = 16
18 plt.rcParams['axes.titlesize'] = 20
19 plt.rcParams['font.size'] = 10
20 plt.rcParams['lines.linewidth'] = 2.0
21 plt.rcParams['lines.markersize'] = 6
22 plt.rcParams['legend.fontsize'] = 14
23 plt.rcParams['axes.facecolor'] = 'white'
24 plt.rcParams['savefig.facecolor'] = 'white'
25 matplotlib.rcParams['xtick.direction'] = 'out'
26 matplotlib.rcParams['ytick.direction'] = 'out'
27 #This import is necessary for the isometric plot.
28 from mpl_toolkits.mplot3d import Axes3D
29 #####
30 N_ = 5 #Choose a number of bodies to explore between 2 and 5.
31 #Choose a stepping method. Choices are:
32 #   'euler' (Euler)
33 #   'rk2'   (2nd Order Runge-Kutta)
34 #   'ec'    (Euler-Cromer)
35 #   'vv'    (velocity Verlet)
36 #   'pv'    (position Verlet)
37 #   'vefrl' (velocity extended Forest-Ruth-like)
38 #   'pefrl' (position extended Forest-Ruth-like)
39 method_ = 'pefrl'
40
41 #####
42 G = 6.67e-11 #universal gravitational constant in units of m^3/kg*s^2
43 end_time = 100*365.26*24*3600 # Define the end time in seconds (e.g. 100 years).
44 dt = 2.0*3600 #Define the length of each time step in seconds (e.g. 2 hours).
45 steps = int(end_time/dt) #Define the total number of time steps.
46 #Fill out a list of times by successively adding increments of dt.
47 t = dt*np.array(range(steps + 1))
48
49 #Define the masses for a number of bodies (in kg).
50 m0 = 2.0e30 #mass of the sun
51 m1 = 3.285e23 #mass of Mercury
52 m2 = 4.8e24 #mass of Venus
53 m3 = 6.0e24 #mass of Earth
54 m4 = 2.4e24 #mass of Mars
55
56 m = [m0, m1, m2, m3, m4] #Combine the masses into an ordered list.
57 #Set colors for the bodies.
58 c0 = '#ff0000' #red
59 c1 = '#8c8c94' #gray
60 c2 = '#ffd56f' #pale yellow
61 c3 = '#005e7b' #sea blue
62 c4 = '#a06534' #reddish brown
63 c = [c0, c1, c2, c3, c4] #Put colors into a similarly ordered list.
64 #####
65 #Orbit takes in a list of masses (floats), the number of bodies N (integer
66 #between 2 and 5), and the desired stepping method (string). This function lays
67 #a foundation for a general N-body function, but such a function's development

```

```

68 #will be saved for later if at all. Five bodies should be enough to get a
69 #general feel for how the problem would extend to N bodies.
70 def orbit(m, N, method):
71     #If you print out either r or v below, you'll see several "layers" of 3x3
72     #matrices. Which layer you are on represents which time step you are on.
73     #Within each 3x3 matrix, the row denotes which body, while the columns 1-3
74     #represent x-, y-, z-positions respectively. In essence, each number in r
75     #or v is associated with three indices: step #, body #, and coordinate #.
76     r = np.zeros([steps+1, N, 3])
77     v = np.zeros([steps+1, N, 3])
78     if N == 2:
79         #Next, we input initial positions. Note the first bracketed triplet of
80         #data represents the x-, y-, and z-position of the first body (m0).
81         r[0] = np.array([[1.0, 3.0, 2.0],
82                         [6.0, -5.0, 4.0]])*1e11
83         #Input initial velocities. Note the sun has zero velocity here.
84         v[0] = np.array([[0.0, 0.0, 0.0],
85                         [7.0, 0.5, 2.0]])*1e3
86     if N == 3:
87         r[0] = np.array([[1.0, 3.0, 2.0],
88                         [6.0, -5.0, 4.0],
89                         [7.0, 8.0, -7.0]])*1e11
90         v[0] = np.array([[0.0, 0.0, 0.0],
91                         [7.0, 0.5, 2.0],
92                         [-4.0, -0.5, -3.0]])*1e3
93     if N == 4:
94         r[0] = np.array([[1.0, 3.0, 2.0],
95                         [6.0, -5.0, 4.0],
96                         [7.0, 8.0, -7.0],
97                         [8.0, 9.0, -6.0]])*1e11
98         v[0] = np.array([[0.0, 0.0, 0.0],
99                         [7.0, 0.5, 2.0],
100                        [-4.0, -0.5, -3.0],
101                        [7.0, 0.5, 2.0]])*1e3
102     if N == 5:
103         r[0] = np.array([[1.0, 3.0, 2.0],
104                         [6.0, -5.0, 4.0],
105                         [7.0, 8.0, -7.0],
106                         [8.0, 9.0, -6.0],
107                         [8.8, 9.8, -6.8]])*1e11
108         v[0] = np.array([[0.0, 0.0, 0.0],
109                         [7.0, 0.5, 2.0],
110                         [-4.0, -0.5, -3.0],
111                         [7.0, 0.5, 2.0],
112                         [7.8, 1.3, 2.8]])*1e3
113
114     #Each body's acceleration at each time step has to do with the force from
115     #all other bodies and vice versa. It can be seen that this formula is
116     #generalizable, but doing so is beyond the scope of this exploration.
117     #See: https://en.wikipedia.org/wiki/N-body\_problem
118     def accel(r):
119         a = np.zeros([N,3])
120         if N == 2:
121             a[0] = G*(m[1]/LA.norm(r[0]-r[1])**3*(r[1]-r[0]))
122             a[1] = G*(m[0]/LA.norm(r[1]-r[0])**3*(r[0]-r[1]))
123         if N == 3:
124             a[0] = G*(m[1]/LA.norm(r[0]-r[1])**3*(r[1]-r[0])
125                     + m[2]/LA.norm(r[0]-r[2])**3*(r[2]-r[0]))
126
127             a[1] = G*(m[0]/LA.norm(r[1]-r[0])**3*(r[0]-r[1])
128                     + m[2]/LA.norm(r[1]-r[2])**3*(r[2]-r[1]))
129
130             a[2] = G*(m[0]/LA.norm(r[2]-r[0])**3*(r[0]-r[2])
131                     + m[1]/LA.norm(r[2]-r[1])**3*(r[1]-r[2]))
132         if N == 4:
133             a[0] = G*(m[1]/LA.norm(r[0]-r[1])**3*(r[1]-r[0])
134                     + m[2]/LA.norm(r[0]-r[2])**3*(r[2]-r[0])

```

```

135         + m[3]/LA.norm(r[0]-r[3])**3*(r[3]-r[0]))
136
137     a[1] = G*(m[0]/LA.norm(r[1]-r[0])**3*(r[0]-r[1])
138         + m[2]/LA.norm(r[1]-r[2])**3*(r[2]-r[1])
139         + m[3]/LA.norm(r[1]-r[3])**3*(r[3]-r[1]))
140
141     a[2] = G*(m[0]/LA.norm(r[2]-r[0])**3*(r[0]-r[2])
142         + m[1]/LA.norm(r[2]-r[1])**3*(r[1]-r[2])
143         + m[3]/LA.norm(r[2]-r[3])**3*(r[3]-r[2]))
144
145     a[3] = G*(m[0]/LA.norm(r[3]-r[0])**3*(r[0]-r[3])
146         + m[1]/LA.norm(r[3]-r[1])**3*(r[1]-r[3])
147         + m[2]/LA.norm(r[3]-r[2])**3*(r[2]-r[3]))
148
149     if N == 5:
150         a[0] = G*(m[1]/LA.norm(r[0]-r[1])**3*(r[1]-r[0])
151             + m[2]/LA.norm(r[0]-r[2])**3*(r[2]-r[0])
152             + m[3]/LA.norm(r[0]-r[3])**3*(r[3]-r[0])
153             + m[4]/LA.norm(r[0]-r[4])**3*(r[4]-r[0]))
154
155         a[1] = G*(m[0]/LA.norm(r[1]-r[0])**3*(r[0]-r[1])
156             + m[2]/LA.norm(r[1]-r[2])**3*(r[2]-r[1])
157             + m[3]/LA.norm(r[1]-r[3])**3*(r[3]-r[1])
158             + m[4]/LA.norm(r[1]-r[4])**3*(r[4]-r[1]))
159
160         a[2] = G*(m[0]/LA.norm(r[2]-r[0])**3*(r[0]-r[2])
161             + m[1]/LA.norm(r[2]-r[1])**3*(r[1]-r[2])
162             + m[3]/LA.norm(r[2]-r[3])**3*(r[3]-r[2])
163             + m[4]/LA.norm(r[2]-r[4])**3*(r[4]-r[2]))
164
165         a[3] = G*(m[0]/LA.norm(r[3]-r[0])**3*(r[0]-r[3])
166             + m[1]/LA.norm(r[3]-r[1])**3*(r[1]-r[3])
167             + m[2]/LA.norm(r[3]-r[2])**3*(r[2]-r[3])
168             + m[4]/LA.norm(r[3]-r[4])**3*(r[4]-r[3]))
169
170         a[4] = G*(m[0]/LA.norm(r[4]-r[0])**3*(r[0]-r[4])
171             + m[1]/LA.norm(r[4]-r[1])**3*(r[1]-r[4])
172             + m[2]/LA.norm(r[4]-r[2])**3*(r[2]-r[4])
173             + m[3]/LA.norm(r[4]-r[3])**3*(r[3]-r[4]))
174
175     return a
176
177 #The simplest implementation is the Euler method. Note that this method
178 #does not conserve energy.
179 if method == 'euler':
180     for i in range(steps):
181         r[i+1] = r[i] + dt*v[i]
182         v[i+1] = v[i] + dt*accel(r[i])
183
184 #The next-simplest method is an Euler-Cromer implementation.
185 if method == 'ec':
186     for i in range(steps):
187         r[i+1] = r[i] + dt*v[i]
188         v[i+1] = v[i] + dt*accel(r[i+1])
189
190 #Getting slightly fancier, we employ the 2nd Order Runge-Kutta method.
191 if method == 'rk2':
192     for i in range(steps):
193         v_iphalf = v[i] + accel(r[i])*(dt/2)
194         r_iphalf = r[i] + v[i]*(dt/2)
195         v[i+1] = v[i] + accel(r_iphalf)*dt
196         r[i+1] = r[i] + v_iphalf*dt
197
198 #Here is a velocity Verlet implementation.
199 #See: http://young.physics.ucsc.edu/115/leapfrog.pdf
200 if method == 'vv':
201     for i in range(steps):
202         v_iphalf = v[i] + (dt/2)*accel(r[i])

```

```

202         r[i+1] = r[i] + dt*v_iphalf
203         v[i+1] = v_iphalf + (dt/2)*accel(r[i+1])
204
205     #Next is a position Verlet implementation (found in the same pdf as 'vv').
206     if method == 'pv':
207         for i in range(steps):
208             r_iphalf = r[i] + (dt/2)*v[i]
209             v[i+1] = v[i] + dt*accel(r_iphalf)
210             r[i+1] = r_iphalf + (dt/2)*v[i+1]
211
212     #EFRL refers to an extended Forest-Ruth-like integration algorithm. Below
213     #are three optimization parameters associated with EFRL routines.
214     e = 0.1786178958448091e0
215     l = -0.2123418310626054e0
216     k = -0.6626458266981849e-1
217     #First we do a velocity EFRL implementation (VEFRL).
218     #See: https://arxiv.org/pdf/cond-mat/0110585.pdf
219     if method == 'vefrl':
220         for i in range(steps):
221             v1 = v[i] + accel(r[i])*e*dt
222             r1 = r[i] + v1*(1-2*l)*(dt/2)
223             v2 = v1 + accel(r1)*k*dt
224             r2 = r1 + v2*l*dt
225             v3 = v2 + accel(r2)*(1-2*(k+e))*dt
226             r3 = r2 + v3*l*dt
227             v4 = v3 + accel(r3)*k*dt
228             r[i+1] = r3 + v4*(1-2*l)*(dt/2)
229             v[i+1] = v4 + accel(r[i+1])*e*dt
230
231     #Next is a position EFRL (PEFRL) (found in the same pdf as 'vefrl').
232     if method == 'pefrl':
233         e = 0.1786178958448091e0
234         l = -0.2123418310626054e0
235         k = -0.6626458266981849e-1
236         for i in range(steps):
237             r1 = r[i] + v[i]*e*dt
238             v1 = v[i] + accel(r1)*(1-2*l)*(dt/2)
239             r2 = r1 + v1*k*dt
240             v2 = v1 + accel(r2)*l*dt
241             r3 = r2 + v2*(1-2*(k+e))*dt
242             v3 = v2 + accel(r3)*l*dt
243             r4 = r3 + v3*k*dt
244             v[i+1] = v3 + accel(r4)*(1-2*l)*(dt/2)
245             r[i+1] = r4 + v[i+1]*e*dt
246
247     return r, v
248
249 #####
250 r, v = orbit(m, N_, method_)
251 if N_ == 2:
252     #r, v = two(m, method_)
253     Nstr = '$\mathrm{Two}\backslash$'
254 elif N_ == 3:
255     #r, v = three(m, method_)
256     Nstr = '$\mathrm{Three}\backslash$'
257 elif N_ == 4:
258     #r, v = four(m, method_)
259     Nstr = '$\mathrm{Four}\backslash$'
260 elif N_ == 5:
261     #r, v = five(m, method_)
262     Nstr = '$\mathrm{Five}\backslash$'
263 #####
264 fig1 = plt.figure(1, facecolor='white')
265 ax1 = fig1.add_subplot(1,1,1, projection='3d')
266 plt.title(r'%s'%Nstr+r'$\mathrm{Orbiting}\backslash$ Bodies$', y=1.05)
267 ax1.set_xlabel(r'$\mathrm{x-position}\backslash$ \mathrm{(m)}$', labelpad=10)
268 ax1.set_ylabel(r'$\mathrm{y-position}\backslash$ \mathrm{(m)}$', labelpad=10)

```

```

269 ax1.set_zlabel(r'$\mathrm{z-position}\ \ \mathrm{(m)}$', labelpad=10)
270
271 a_ = 0.7 #Set a transparency value so we can see where orbits overlap.
272 #For all times, plot (x,y,z) tuples for m0.
273 ax1.plot(r[:, 0, 0], r[:, 0, 1], r[:, 0, 2], color=c[0], label=r'$m_0$',
274         alpha=a_)
275 #For all times, plot (x,y,z) tuples for m1, etc.
276 ax1.plot(r[:, 1, 0], r[:, 1, 1], r[:, 1, 2], color=c[1], label=r'$m_1$',
277         alpha=a_)
278 if N_ > 2:
279     ax1.plot(r[:, 2, 0], r[:, 2, 1], r[:, 2, 2], color=c[2], label=r'$m_2$',
280             alpha=a_)
281 if N_ > 3:
282     ax1.plot(r[:, 3, 0], r[:, 3, 1], r[:, 3, 2], color=c[3], label=r'$m_3$',
283             alpha=a_)
284 if N_ > 4:
285     ax1.plot(r[:, 4, 0], r[:, 4, 1], r[:, 4, 2], color=c[4], label=r'$m_4$',
286             alpha=a_)
287 ax1.axis('equal')
288 plt.legend(loc='upper left')
289 #####
290 fig2 = plt.figure(2, facecolor='white')
291 ax2 = fig2.add_subplot(111)
292 plt.title(r'%s'%Nstr
293          +r'$\mathrm{Orbiting\ Bodies\ }$'+'\n'
294          +r'$\mathrm{as\ Viewed\ From\ the\ Positive\ x-Axis}$', y=1.05)
295 ax2.set_xlabel(r'$\mathrm{y-position}\ \ \mathrm{(m)}$')
296 ax2.set_ylabel(r'$\mathrm{z-position}\ \ \mathrm{(m)}$')
297 #For all times, plot (x,z) tuples for m0.
298 ax2.plot(r[:, 0, 1], r[:, 0, 2], color=c[0], label=r'$m_0$', alpha=a_)
299 #For all times, plot (x,z) tuples for m1, etc.
300 ax2.plot(r[:, 1, 1], r[:, 1, 2], color=c[1], label=r'$m_1$', alpha=a_)
301 if N_ > 2:
302     ax2.plot(r[:, 2, 1], r[:, 2, 2], color=c[2], label=r'$m_2$', alpha=a_)
303 if N_ > 3:
304     ax2.plot(r[:, 3, 1], r[:, 3, 2], color=c[3], label=r'$m_3$', alpha=a_)
305 if N_ > 4:
306     ax2.plot(r[:, 4, 1], r[:, 4, 2], color=c[4], label=r'$m_4$', alpha=a_)
307 ax2.axis('equal')
308 ax2.legend(loc='lower right')
309 #####
310 fig3 = plt.figure(3, facecolor='white')
311 ax3 = fig3.add_subplot(111)
312 plt.title(r'%s'%Nstr
313          +r'$\mathrm{Orbiting\ Bodies\ }$'+'\n'
314          +r'$\mathrm{as\ Viewed\ From\ the\ Positive\ y-Axis}$', y=1.05)
315 ax3.set_xlabel(r'$\mathrm{x-position}\ \ \mathrm{(m)}$')
316 ax3.set_ylabel(r'$\mathrm{z-position}\ \ \mathrm{(m)}$')
317 #For all times, plot (x,z) tuples for m0.
318 ax3.plot(r[:, 0, 0], r[:, 0, 2], color=c[0], label=r'$m_0$', alpha=a_)
319 #For all times, plot (x,z) tuples for m1, etc.
320 ax3.plot(r[:, 1, 0], r[:, 1, 2], color=c[1], label=r'$m_1$', alpha=a_)
321 if N_ > 2:
322     ax3.plot(r[:, 2, 0], r[:, 2, 2], color=c[2], label=r'$m_2$', alpha=a_)
323 if N_ > 3:
324     ax3.plot(r[:, 3, 0], r[:, 3, 2], color=c[3], label=r'$m_3$', alpha=a_)
325 if N_ > 4:
326     ax3.plot(r[:, 4, 0], r[:, 4, 2], color=c[4], label=r'$m_4$', alpha=a_)
327 ax3.axis('equal')
328 ax3.legend(loc='lower right')
329 #####
330 fig4 = plt.figure(4, facecolor='white')
331 ax4 = fig4.add_subplot(111)
332 plt.title(r'%s'%Nstr
333          +r'$\mathrm{Orbiting\ Bodies\ }$'+'\n'
334          +r'$\mathrm{as\ Viewed\ From\ the\ Positive\ z-Axis}$', y=1.05)
335 ax4.set_xlabel(r'$\mathrm{x-position}\ \ \mathrm{(m)}$')

```

```

336 ax4.set_ylabel(r'$\mathrm{y-position} \setminus \mathrm{(m)}$')
337 #For all times, plot (x,z) tuples for m0.
338 ax4.plot(r[:, 0, 0], r[:, 0, 1], color=c[0], label=r'$m_0$', alpha=a_)
339 #For all times, plot (x,z) tuples for m1, etc.
340 ax4.plot(r[:, 1, 0], r[:, 1, 1], color=c[1], label=r'$m_1$', alpha=a_)
341 if N_ > 2:
342     ax4.plot(r[:, 2, 0], r[:, 2, 1], color=c[2], label=r'$m_2$', alpha=a_)
343 if N_ > 3:
344     ax4.plot(r[:, 3, 0], r[:, 3, 1], color=c[3], label=r'$m_3$', alpha=a_)
345 if N_ > 4:
346     ax4.plot(r[:, 4, 0], r[:, 4, 1], color=c[4], label=r'$m_4$', alpha=a_)
347 ax4.axis('equal')
348 ax4.legend(loc='lower right')

```