

# Introduction to Computational Physics - Exercise 3

Simon Groß-Bölting, Lorenz Vogel, Sebastian Willenberg

15. Mai 2020

## 2 Three-Body Problem

The Runge-Kutta-4 integrator was adapted to the gravitational 3-body problem. The system was simplified by setting the gravitational constant to  $G = 1$  and the dimensionality was reduced to a two dimensional plane. The system was configured with the following initial conditions.

- (a) In a first step, the masses of all three bodies were set to  $m_1 = m_2 = m_3 = 1$  and the following initial conditions were selected for  $y(0)$ :

$$\begin{aligned}(y_1, y_2) &= -0.97000436; 0.24308753 \\(y_3, y_4) &= -0.46620368; -0.43236573 \\(y_5, y_6) &= 0.97000436; -0.24308753 \\(y_7, y_8) &= -0.46620368; -0.43236573 \\(y_9, y_{10}) &= 0.0; 0.0 \\(y_{11}, y_{12}) &= 0.93240737; 0.86473146\end{aligned}$$

Here,  $y_{1+4i}$  and  $y_{2+4i}$  are the initial coordinates and  $y_{3+4i}$  and  $y_{4+4i}$  the initial velocities for the objects  $i = 0, 1$  and  $2$ . The integration was done with a step size  $h$  between  $0.01$  and  $0.001$ :

```
1 # -*- coding: utf-8 -*-
2 """
3 Introduction to Computational Physics
4 - Exercise 03: Numerical Simulation of the Three-Body Problem
5               using the 4th Order Runge-Kutta Method
6 - Group: Simon Groß-Bölting, Lorenz Vogel, Sebastian Willenberg
7 """
8
9 import numpy as np
10 import matplotlib.pyplot as plt
11 import matplotlib.animation as animation
12
13 def rk4_step(y0, x0, f, h, f_args = {}):
14     ''' Simple python implementation for one RK4 step.
15     Inputs:
16         y_0      - M x 1 numpy array specifying all variables of the ODE
17                   at the current time step
18         x_0      - current time step
19         f        - function that calculates the derivatives of all
20                   variables of the ODE
21         h        - time step size
22         f_args   - Dictionary of additional arguments to be passed to the
23                   function f
24     Output:
```

```

22         yp1 - M x 1 numpy array of variables at time step x0 + h
23         xp1 - time step x0+h
24     '''
25     k1 = h * f(y0, x0, **f_args)
26     k2 = h * f(y0 + k1/2., x0 + h/2., **f_args)
27     k3 = h * f(y0 + k2/2., x0 + h/2., **f_args)
28     k4 = h * f(y0 + k3, x0 + h, **f_args)
29
30     xp1 = x0 + h
31     yp1 = y0 + 1./6.*(k1 + 2.*k2 + 2.*k3 + k4)
32
33     return(yp1,xp1)
34
35 def rk4(y0, x0, f, h, n, f_args = {}):
36     ''' Simple implementation of RK4
37         Inputs:
38             y_0      - M x 1 numpy array specifying all variables of the ODE
39             at the current time step
40             x_0      - current time step
41             f         - function that calculates the derivatives of all
42             variables of the ODE
43             h         - time step size
44             n         - number of steps
45             f_args    - Dictionary of additional arguments to be passed to the
46             function f
47         Output:
48             yn - N+1 x M numpy array with the results of the integration
49             for every time step (includes y0)
50             xn - N+1 x 1 numpy array with the time step value (includes
51             start x0)
52         '''
53         yn = np.zeros((n+1, y0.shape[0]))
54         xn = np.zeros(n+1)
55         yn[0,:] = y0
56         xn[0] = x0
57
58         for n in np.arange(1,n+1,1):
59             yn[n,:], xn[n] = rk4_step(y0 = yn[n-1,:], x0 = xn[n-1], f = f, h =
60             h, f_args = f_args)
61
62         return(yn, xn)
63
64 def three_body_problem(y,x,G,m1,m2,m3):
65     ''' Twelve coupled ordinary differential equations of first order
66         (converted into the standard form) '''
67     yn = np.ones(12)
68     x1 = y[0]; y1 = y[1]
69     x2 = y[4]; y2 = y[5]
70     x3 = y[8]; y3 = y[9]
71
72     r12 = np.sqrt((x1-x2)**2+(y1-y2)**2) # distance between body 1 and body
73     2
74     r13 = np.sqrt((x1-x3)**2+(y1-y3)**2) # distance between body 1 and body
75     3
76     r23 = np.sqrt((x2-x3)**2+(y2-y3)**2) # distance between body 2 and body
77     3
78
79     yn[0] = y[2] # differential equations for body 1
80     yn[1] = y[3]
81     yn[2] = (-m2*G/r12**3)*(x1-x2)+(-m3*G/r13**3)*(x1-x3)
82     yn[3] = (-m2*G/r12**3)*(y1-y2)+(-m3*G/r13**3)*(y1-y3)
83
84     yn[4] = y[6] # differential equations for body 2

```

```

76     yn[5] = y[7]
77     yn[6] = (-m1*G/r12**3)*(x2-x1)+(-m3*G/r23**3)*(x2-x3)
78     yn[7] = (-m1*G/r12**3)*(y2-y1)+(-m3*G/r23**3)*(y2-y3)
79
80     yn[8] = y[10]    # differential equations for body 3
81     yn[9] = y[11]
82     yn[10] = (-m1*G/r13**3)*(x3-x1)+(-m2*G/r23**3)*(x3-x2)
83     yn[11] = (-m1*G/r13**3)*(y3-y1)+(-m2*G/r23**3)*(y3-y2)
84
85     return yn
86
87 class Body:
88     def __init__(self,mass,position,velocity=np.array([0.,0.])):
89         self.mass = mass          # mass of the body
90         self.position = position   # initial position vector
91         self.velocity = velocity   # initial velocity vector
92
93 def initial_conditions(body1,body2,body3):
94     ''' Function to write the initial conditions into an numpy array '''
95     return np.array([body1.position[0],body1.position[1],
96                     body1.velocity[0],body1.velocity[1],
97                     body2.position[0],body2.position[1],
98                     body2.velocity[0],body2.velocity[1],
99                     body3.position[0],body3.position[1],
100                    body3.velocity[0],body3.velocity[1]])
101
102
103 G = 1. # simplify the system by setting the gravitational constant to G=1
104 # create the three bodies with their initial conditions
105
106 body1 = Body(1., np.array([-0.97000436,0.24308753]), np.array
107            ([-0.46620368,-0.43236573]))
108 body2 = Body(1., np.array([0.97000436,-0.24308753]), np.array
109            ([-0.46620368,-0.43236573]))
110 body3 = Body(1., np.array([0.,0.]), np.array([0.93240737,0.86473146]))
111
112 #body1 = Body(1., np.array([0.,0.]), np.array([0.,1.]))
113 #body2 = Body(1., np.array([1.,0.]))
114 #body3 = Body(1., np.array([-1.,0.]), np.array([0.,-1.]))
115
116 #body1 = Body(3., np.array([1.,3.]))
117 #body2 = Body(4., np.array([-2.,-1.]))
118 #body3 = Body(5., np.array([1.,-1.]))
119
120 #body1 = Body(3., np.array([3.,1.]))
121 #body2 = Body(4., np.array([-2.,-1.]))
122 #body3 = Body(5., np.array([1.,-1.]),np.array([0.1,0]))
123
124 # numerical simulation of the gravitational three-body problem
125 # using the Runge-Kutta-4 integrator
126 yn, xn = rk4(initial_conditions(body1,body2,body3),0,three_body_problem,1e
127             -3,int(6320),
128             {'G':G, 'm1':body1.mass, 'm2':body2.mass, 'm3':body3.mass})
129
130 # plot the animated trajectories of the three bodies
131 fig, ax = plt.subplots()
132
133 ax.set_title('Numerical Simulation of the Gravitational Three-Body Problem')
134 ax.set_xlabel(r'$x$-coordinates'); ax.set_ylabel(r'$y$-coordinates')
135
136 ax.plot(yn[:,0], yn[:,1], color='black', ls='-', lw=1)
137 line1, = ax.plot([], [], 'r.', ms=40, label='Body 1')

```

```

135 line2, = ax.plot([], [], 'g.', ms=40, label='Body 2')
136 line3, = ax.plot([], [], 'b.', ms=40, label='Body 3')
137
138 ax.legend(loc='upper right', markerscale=0.4)
139 ax.set_xlim((-1.2,1.2)); ax.set_ylim((-1,1))
140
141 def trajectories(i):
142     index = i*10
143     line1.set_data(yn[index-1:index,0], yn[index-1:index,1])
144     line2.set_data(yn[index-1:index,4], yn[index-1:index,5])
145     line3.set_data(yn[index-1:index,8], yn[index-1:index,9])
146     return (line1, line2, line3,)
147
148 animate = animation.FuncAnimation(fig, trajectories, frames=int(6320/10),
149                                   interval=1, repeat=True, blit=True)
150 plt.show(); plt.clf(); plt.close()
151
152

```

The results of this exercise are in the attached folder ('figures').

- (b) In this part of the exercise we tried to solve the Meissel-Burrau problem. In our solution for the three body problem, the masses of the three bodies were to  $m_1 = 3, m_2 = 4$  and  $m_3 = 5$ . The bodies were placed at the corners of a right triangle (one angle is  $90^\circ$ ) with edge lengths of  $l_1 = 3, l_2 = 4$  and  $l_3 = 5$ , such that  $m_1$  is opposite to the edge  $l_1$ ,  $m_2$  opposite to  $l_2$ , and  $m_3$  opposite to  $l_3$ . The initial velocities were set to zero. The origin of the coordinate system was set into the center of mass of the system.

```

1  # -*- coding: utf-8 -*-
2  """
3  Introduction to Computational Physics
4  - Exercise 03: Numerical Simulation of the Three-Body Problem
5                  using the 4th Order Runge-Kutta Method
6  - Group: Simon Groß-Bölting, Lorenz Vogel, Sebastian Willenberg
7  """
8
9  import numpy as np
10 import matplotlib.pyplot as plt
11 from scipy.signal import argrelextrema
12
13 def rk4_step(y0, x0, f, h, f_args = {}):
14     ''' Simple python implementation for one RK4 step.
15         Inputs:
16             y_0      - M x 1 numpy array specifying all variables of the ODE
17                       at the current time step
18             x_0      - current time step
19             f         - function that calculates the derivatives of all
20                       variables of the ODE
21             h         - time step size
22             f_args    - Dictionary of additional arguments to be passed to the
23                       function f
24         Output:
25             yp1 - M x 1 numpy array of variables at time step x0 + h
26             xp1 - time step x0+h
27     '''
28     k1 = h * f(y0, x0, **f_args)
29     k2 = h * f(y0 + k1/2., x0 + h/2., **f_args)
30     k3 = h * f(y0 + k2/2., x0 + h/2., **f_args)
31     k4 = h * f(y0 + k3, x0 + h, **f_args)
32
33     xp1 = x0 + h
34     yp1 = y0 + 1./6.*(k1 + 2.*k2 + 2.*k3 + k4)
35

```

```

33     return(y0, x0)
34
35 def rk4(y0, x0, f, h, n, f_args = {}):
36     ''' Simple implementation of RK4
37         Inputs:
38             y_0      - M x 1 numpy array specifying all variables of the ODE
39             x_0      - current time step
40             f         - function that calculates the derivatives of all
41             variables of the ODE
42             h         - time step size
43             n         - number of steps
44             f_args    - Dictionary of additional arguments to be passed to the
45             function f
46         Output:
47             yn - N+1 x M numpy array with the results of the integration
48             for every time step (includes y0)
49             xn - N+1 x 1 numpy array with the time step value (includes
50             start x0)
51         '''
52         yn = np.zeros((n+1, y0.shape[0]))
53         xn = np.zeros(n+1)
54         yn[0,:] = y0
55         xn[0] = x0
56
57         for n in np.arange(1,n+1,1):
58             yn[n,:], xn[n] = rk4_step(y0 = yn[n-1,:], x0 = xn[n-1], f = f, h =
59             h, f_args = f_args)
60
61     return(yn, xn)
62
63 def three_body_problem(y,x,G,m1,m2,m3):
64     ''' Twelve coupled ordinary differential equations of first order
65     (converted into the standard form) '''
66     yn = np.ones(12)
67     x1 = y[0]; y1 = y[1]
68     x2 = y[4]; y2 = y[5]
69     x3 = y[8]; y3 = y[9]
70
71     r12 = np.sqrt((x1-x2)**2+(y1-y2)**2) # distance between body 1 and body
72     2
73     r13 = np.sqrt((x1-x3)**2+(y1-y3)**2) # distance between body 1 and body
74     3
75     r23 = np.sqrt((x2-x3)**2+(y2-y3)**2) # distance between body 2 and body
76     3
77
78     yn[0] = y[2] # differential equations for body 1
79     yn[1] = y[3]
80     yn[2] = (-m2*G/r12**3)*(x1-x2)+(-m3*G/r13**3)*(x1-x3)
81     yn[3] = (-m2*G/r12**3)*(y1-y2)+(-m3*G/r13**3)*(y1-y3)
82
83     yn[4] = y[6] # differential equations for body 2
84     yn[5] = y[7]
85     yn[6] = (-m1*G/r12**3)*(x2-x1)+(-m3*G/r23**3)*(x2-x3)
86     yn[7] = (-m1*G/r12**3)*(y2-y1)+(-m3*G/r23**3)*(y2-y3)
87
88     yn[8] = y[10] # differential equations for body 3
89     yn[9] = y[11]
90     yn[10] = (-m1*G/r13**3)*(x3-x1)+(-m2*G/r23**3)*(x3-x2)
91     yn[11] = (-m1*G/r13**3)*(y3-y1)+(-m2*G/r23**3)*(y3-y2)
92
93     return yn
94
95

```

```

87 class Body:
88     def __init__(self, mass, position, velocity=np.array([0., 0.])):
89         self.mass = mass          # mass of the body
90         self.position = position   # initial position vector
91         self.velocity = velocity   # initial velocity vector
92
93 def initial_conditions(body1, body2, body3):
94     ''' Function to write the initial conditions into an numpy array '''
95     return np.array([body1.position[0], body1.position[1],
96                     body1.velocity[0], body1.velocity[1],
97                     body2.position[0], body2.position[1],
98                     body2.velocity[0], body2.velocity[1],
99                     body3.position[0], body3.position[1],
100                     body3.velocity[0], body3.velocity[1]])
101
102 def min_separation(yn, xn):
103     ''' Function to compute the separation between two bodies
104         and store this data in a file '''
105     separation = np.zeros((len(xn), 4))
106     separation[:, 0] = xn          # time column
107     # compute the distance between two bodies
108     separation[:, 1] = np.sqrt((yn[:, 0] - yn[:, 4])**2 + (yn[:, 1] - yn[:, 5])**2)
109     separation[:, 2] = np.sqrt((yn[:, 0] - yn[:, 8])**2 + (yn[:, 1] - yn[:, 9])**2)
110     separation[:, 3] = np.sqrt((yn[:, 4] - yn[:, 8])**2 + (yn[:, 5] - yn[:, 9])**2)
111
112     # find the minimum distances between two bodies
113     index_min_12 = argrelextrema(separation[:, 1], np.less)[0]
114     index_min_13 = argrelextrema(separation[:, 2], np.less)[0]
115     index_min_23 = argrelextrema(separation[:, 3], np.less)[0]
116
117     min_separation_12 = np.array([separation[:, 0][index_min_12],
118                                 separation[:, 1][index_min_12]])
119     min_separation_13 = np.array([separation[:, 0][index_min_13],
120                                 separation[:, 2][index_min_13]])
121     min_separation_23 = np.array([separation[:, 0][index_min_23],
122                                 separation[:, 3][index_min_23]])
123
124     # store the results into txt-files
125     np.savetxt('data/separation.txt', separation, delimiter='\t')
126     np.savetxt('data/min_separation_12.txt', min_separation_12, delimiter=
127     '\t')
128     np.savetxt('data/min_separation_13.txt', min_separation_13, delimiter=
129     '\t')
130     np.savetxt('data/min_separation_23.txt', min_separation_23, delimiter=
131     '\t')
132
133 def error_total_energy(G, body1, body2, body3, yn):
134     ''' Function to compute the relative error of the total energy
135         of the system compared to the initial value '''
136
137     # compute the total kinetic energy of the system
138     kin_energy = 0.5*(body1.mass*(yn[:, 2]**2 + yn[:, 3]**2)
139                     + body2.mass*(yn[:, 6]**2 + yn[:, 7]**2)
140                     + body3.mass*(yn[:, 10]**2 + yn[:, 11]**2))
141
142     # compute the total potential energy of the system
143     r12 = np.sqrt((yn[:, 0] - yn[:, 4])**2 + (yn[:, 1] - yn[:, 5])**2)
144     r13 = np.sqrt((yn[:, 0] - yn[:, 8])**2 + (yn[:, 1] - yn[:, 9])**2)
145     r23 = np.sqrt((yn[:, 4] - yn[:, 8])**2 + (yn[:, 5] - yn[:, 9])**2)
146     pot_energy = -G*((body1.mass*body2.mass/r12)
147                    + (body1.mass*body3.mass/r13)
148                    + (body2.mass*body3.mass/r23))

```

```

147     # compute the total energy of the system and the relative error
148     total_energy = kin_energy+pot_energy
149     relative_error = abs(total_energy-total_energy[0])/abs(total_energy[0])
150     return relative_error
151
152
153 G = 1. # simplify the system by setting the gravitational constant to G=1
154 # create the three bodies with their initial conditions:
155 # Meissel-Burrau problem or Pythagorean problem
156 body1 = Body(3., np.array([1.,3.]))
157 body2 = Body(4., np.array([-2.,-1.]))
158 body3 = Body(5., np.array([1.,-1.]))
159
160 # numerical simulation of the gravitational three-body problem
161 # using the Runge-Kutta-4 integrator
162 yn, xn = rk4(initial_conditions(body1,body2,body3),0,three_body_problem,4e
    -5,int(5e5),
163             {'G':G, 'm1':body1.mass, 'm2':body2.mass, 'm3':body3.mass})
164
165
166 min_separation(yn,xn) # compute the minimum separation
167 separation = np.loadtxt('data/separation.txt') # load distance data from
    files
168 min_sep_12 = np.loadtxt('data/min_separation_12.txt')
169 min_sep_13 = np.loadtxt('data/min_separation_13.txt')
170 min_sep_23 = np.loadtxt('data/min_separation_23.txt')
171
172 # compute the total energy and the relative error of the total energy
173 relative_error = error_total_energy(G,body1,body2,body3,yn)
174
175
176 # plot the trajectories of the three bodies in the orbital plane
177 fig, ax = plt.subplots()
178 ax.set_title('Numerical Simulation of the Gravitational Three-Body Problem\
    n'+
179             'trajectories of the three bodies in the orbital plane')
180 ax.set_xlabel(r'$x$-coordinates'); ax.set_ylabel(r'$y$-coordinates')
181 ax.plot(yn[:,0], yn[:,1], 'r.', markersize=1, label='Body 1')
182 ax.plot(yn[:,4], yn[:,5], 'g.', markersize=1, label='Body 2')
183 ax.plot(yn[:,8], yn[:,9], 'b.', markersize=1, label='Body 3')
184 ax.set_xlim((-3.5,3.5)); ax.set_ylim((-3,4))
185 ax.grid(); ax.legend(loc='best', markerscale=8)
186 fig.savefig('figures/Meissel-Burrau-Trajectories.png', format='png')
187
188 # plot the time evolution of the distance between two bodies
189 fig, ax = plt.subplots()
190 ax.set_title('Numerical Simulation of the Gravitational Three-Body Problem\
    n'+
191             'time evolution of the distance between two bodies')
192 ax.set_xlabel('time'); ax.set_ylabel('distance')
193 ax.plot(separation[:,0], separation[:,1], 'r.', markersize=1, label='Bodies
    1 and 2')
194 ax.plot(separation[:,0], separation[:,2], 'g.', markersize=1, label='Bodies
    1 and 3')
195 ax.plot(separation[:,0], separation[:,3], 'b.', markersize=1, label='Bodies
    2 and 3')
196 ax.grid(); ax.legend(loc='best', markerscale=8); ax.set_yscale('log')
197 fig.savefig('figures/Meissel-Burrau-Distances.png', format='png')
198
199 # plot the time evolution of the minimum separation between two bodies
200 fig, ax = plt.subplots()
201 ax.set_title('Numerical Simulation of the Gravitational Three-Body Problem\
    n'+

```

```

202         'time evolution of the minimum separation between two bodies')
203 ax.set_xlabel('time'); ax.set_ylabel('distance')
204 ax.plot(min_sep_12[0], min_sep_12[1], 'rx', label='Bodies 1 and 2')
205 ax.plot(min_sep_13[0], min_sep_13[1], 'gx', label='Bodies 1 and 3')
206 ax.plot(min_sep_23[0], min_sep_23[1], 'bx', label='Bodies 2 and 3')
207 ax.grid(); ax.legend(loc='best'); ax.set_yscale('log')
208 fig.savefig('figures/Meissel-Burrau_Minimum-Separation.png', format='png')
209
210 # plot the time evolution of the relative error of the total energy
211 fig, ax = plt.subplots()
212 ax.set_title('Numerical Simulation of the Gravitational Three-Body Problem\
n'+
213             'time evolution of the relative error of the total energy')
214 ax.set_xlabel('time'); ax.set_ylabel('relative error of the total energy')
215 ax.plot(xn, relative_error, 'b.', markersize=1); ax.grid(); ax.set_yscale('
log')
216 fig.savefig('figures/Meissel-Burrau_Realtive-Error.png', format='png')
217
218 plt.show(); plt.clf(); plt.close()

```

The results of this exercise are in the attached folder ('figures').

- (c) In this part of the exercise the initial configuration was set to be the same as in (b). The initial velocity was set to  $v = 0.1$  for the most massive particle ( $m_3 = 5$ ) in the direction of the body  $m_2 = 4$ .

The results of this exercise are in the attached folder ('figures').