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 adaptet to the gravitational 3-body problem. The System was symplified by setting the gravitational constant to $G = 1$ and the diemensionality was reduced to a two dimentional plane. The system was configured with the following initial conditions.

(a) In a rst step, the masses of all three bodies where set to $m_1 = m_2 = m_3 = 1$ and the following initial conditions where 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:

```
# -*- coding: utf-8 -*-
"""
Introduction to Computational Physics
- Exercise 03:  Numerical Simulation of the Three-Body Problem
                using the 4th Order Runge-Kutta Method
- Group: Simon Groß-Bölting, Lorenz Vogel, Sebastian Willenberg
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def rk4_step(y0, x0, f, h, f_args = {}):
    ''' Simple python implementation for one RK4 step.
        Inputs:
            y_0    - M x 1 numpy array specifying all variables of the ODE at the current time step
            x_0    - current time step
            f      - function that calculates the derivates of all variables of the ODE
            h      - time step size
            f_args - Dictionary of additional arguments to be passed to the function f
        Output:
```

```
            yp1 - M x 1 numpy array of variables at time step x0 + h
            xp1 - time step x0+h
    '''
    k1 = h * f(y0, x0, **f_args)
    k2 = h * f(y0 + k1/2., x0 + h/2., **f_args)
    k3 = h * f(y0 + k2/2., x0 + h/2., **f_args)
    k4 = h * f(y0 + k3, x0 + h, **f_args)

    xp1 = x0 + h
    yp1 = y0 + 1./6.*(k1 + 2.*k2 + 2.*k3 + k4)

    return(yp1,xp1)

def rk4(y0, x0, f, h, n, f_args = {}):
    ''' Simple implementation of RK4
        Inputs:
            y_0    - M x 1 numpy array specifying all variables of the ODE at the current time step
            x_0    - current time step
            f      - function that calculates the derivates of all variables of the ODE
            h      - time step size
            n      - number of steps
            f_args - Dictionary of additional arguments to be passed to the function f
        Output:
            yn - N+1 x M numpy array with the results of the integration for every time step (includes y0)
            xn - N+1 x 1 numpy array with the time step value (includes start x0)
    '''
    yn = np.zeros((n+1, y0.shape[0]))
    xn = np.zeros(n+1)
    yn[0,:] = y0
    xn[0] = x0

    for n in np.arange(1,n+1,1):
        yn[n,:], xn[n] = rk4_step(y0 = yn[n-1,:], x0 = xn[n-1], f = f, h = h, f_args = f_args)

    return(yn, xn)

def three_body_problem(y,x,G,m1,m2,m3):
    ''' Twelve coupled ordinary differential equations of first order
        (converted into the standard form) '''
    yn = np.ones(12)
    x1 = y[0];  y1 = y[1]
    x2 = y[4];  y2 = y[5]
    x3 = y[8];  y3 = y[9]

    r12 = np.sqrt((x1-x2)**2+(y1-y2)**2) # distance between body 1 and body 2
    r13 = np.sqrt((x1-x3)**2+(y1-y3)**2) # distance between body 1 and body 3
    r23 = np.sqrt((x2-x3)**2+(y2-y3)**2) # distance between body 2 and body 3

    yn[0] = y[2]    # differential equations for body 1
    yn[1] = y[3]
    yn[2] = (-m2*G/r12**3)*(x1-x2)+(-m3*G/r13**3)*(x1-x3)
    yn[3] = (-m2*G/r12**3)*(y1-y2)+(-m3*G/r13**3)*(y1-y3)

    yn[4] = y[6]    # differential equations for body 2
```

```
    yn[5] = y[7]
    yn[6] = (-m1*G/r12**3)*(x2-x1)+(-m3*G/r23**3)*(x2-x3)
    yn[7] = (-m1*G/r12**3)*(y2-y1)+(-m3*G/r23**3)*(y2-y3)

    yn[8] = y[10]   # differential equations for body 3
    yn[9] = y[11]
    yn[10] = (-m1*G/r13**3)*(x3-x1)+(-m2*G/r23**3)*(x3-x2)
    yn[11] = (-m1*G/r13**3)*(y3-y1)+(-m2*G/r23**3)*(y3-y2)

    return yn

class Body:
    def __init__(self,mass,position,velocity=np.array([0.,0.])):
        self.mass = mass              # mass of the body
        self.position = position     # inital position vector
        self.velocity = velocity     # inital velocity vector

def initial_conditions(body1,body2,body3):
    ''' Function to write the inital conditions into an numpy array '''
    return np.array([body1.position[0],body1.position[1],
                     body1.velocity[0],body1.velocity[1],
                     body2.position[0],body2.position[1],
                     body2.velocity[0],body2.velocity[1],
                     body3.position[0],body3.position[1],
                     body3.velocity[0],body3.velocity[1]])


G = 1.  # simplify the system by setting the gravitational constant to G=1
# create the three bodies with their initial conditions

body1 = Body(1., np.array([-0.97000436,0.24308753]), np.array([-0.46620368,-0.43236573]))
body2 = Body(1., np.array([0.97000436,-0.24308753]), np.array([-0.46620368,-0.43236573]))
body3 = Body(1., np.array([0.,0.]), np.array([0.93240737,0.86473146]))

#body1 = Body(1., np.array([0.,0.]), np.array([0.,1.]))
#body2 = Body(1., np.array([1.,0.]))
#body3 = Body(1., np.array([-1.,0.]), np.array([0.,-1.]))

#body1 = Body(3., np.array([1.,3.]))
#body2 = Body(4., np.array([-2.,-1.]))
#body3 = Body(5., np.array([1.,-1.]))

#body1 = Body(3., np.array([3.,1.]))
#body2 = Body(4., np.array([-2.,-1.]))
#body3 = Body(5., np.array([1.,-1.]),np.array([0.1,0]))

# numerical simulation of the gravitational three-body problem
# using the Runge-Kutta-4 integrator
yn, xn = rk4(initial_conditions(body1,body2,body3),0,three_body_problem,1e-3,int(6320),
             {'G':G, 'm1':body1.mass, 'm2':body2.mass, 'm3':body3.mass})

# plot the animated trajectories of the three bodies
fig, ax = plt.subplots()

ax.set_title('Numerical Simulation of the Gravitational Three-Body Problem')
ax.set_xlabel(r'$x$-coordinates'); ax.set_ylabel(r'$y$-coordinates')

ax.plot(yn[:,0], yn[:,1], color='black', ls='-', lw=1)
line1, = ax.plot([], [], 'r.', ms=40, label='Body 1')
```

```python
line2, = ax.plot([], [], 'g.', ms=40, label='Body 2')
line3, = ax.plot([], [], 'b.', ms=40, label='Body 3')

ax.legend(loc='upper right', markerscale=0.4)
ax.set_xlim((-1.2,1.2)); ax.set_ylim((-1,1))

def trajectories(i):
    index = i*10
    line1.set_data(yn[index-1:index,0], yn[index-1:index,1])
    line2.set_data(yn[index-1:index,4], yn[index-1:index,5])
    line3.set_data(yn[index-1:index,8], yn[index-1:index,9])
    return (line1, line2, line3,)

animate = animation.FuncAnimation(fig, trajectories, frames=int(6320/10),
                                  interval=1, repeat=True, blit=True)
plt.show(); plt.clf(); plt.close()


```

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°) with edge lengths of $l_1 = 3, l_2 = 4$ and $l_3 = 5$, such that m_1 is opposite to the edge $l1$, 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.

```python
# -*- coding: utf-8 -*-
"""
Introduction to Computational Physics
- Exercise 03:  Numerical Simulation of the Three-Body Problem
                using the 4th Order Runge-Kutta Method
- Group: Simon Groß-Bölting, Lorenz Vogel, Sebastian Willenberg
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import argrelextrema

def rk4_step(y0, x0, f, h, f_args = {}):
    ''' Simple python implementation for one RK4 step.
        Inputs:
            y_0    - M x 1 numpy array specifying all variables of the ODE at the current time step
            x_0    - current time step
            f      - function that calculates the derivates of all variables of the ODE
            h      - time step size
            f_args - Dictionary of additional arguments to be passed to the function f
        Output:
            yp1 - M x 1 numpy array of variables at time step x0 + h
            xp1 - time step x0+h
    '''
    k1 = h * f(y0, x0, **f_args)
    k2 = h * f(y0 + k1/2., x0 + h/2., **f_args)
    k3 = h * f(y0 + k2/2., x0 + h/2., **f_args)
    k4 = h * f(y0 + k3, x0 + h, **f_args)

    xp1 = x0 + h
    yp1 = y0 + 1./6.*(k1 + 2.*k2 + 2.*k3 + k4)

```

```
    return(yp1,xp1)

def rk4(y0, x0, f, h, n, f_args = {}):
    ''' Simple implementation of RK4
        Inputs:
            y_0    - M x 1 numpy array specifying all variables of the ODE at the current time step
            x_0    - current time step
            f      - function that calculates the derivates of all variables of the ODE
            h      - time step size
            n      - number of steps
            f_args - Dictionary of additional arguments to be passed to the function f
        Output:
            yn - N+1 x M numpy array with the results of the integration for every time step (includes y0)
            xn - N+1 x 1 numpy array with the time step value (includes start x0)
    '''
    yn = np.zeros((n+1, y0.shape[0]))
    xn = np.zeros(n+1)
    yn[0,:] = y0
    xn[0] = x0

    for n in np.arange(1,n+1,1):
        yn[n,:], xn[n] = rk4_step(y0 = yn[n-1,:], x0 = xn[n-1], f = f, h = h, f_args = f_args)

    return(yn, xn)

def three_body_problem(y,x,G,m1,m2,m3):
    ''' Twelve coupled ordinary differential equations of first order
        (converted into the standard form) '''
    yn = np.ones(12)
    x1 = y[0];  y1 = y[1]
    x2 = y[4];  y2 = y[5]
    x3 = y[8];  y3 = y[9]

    r12 = np.sqrt((x1-x2)**2+(y1-y2)**2) # distance between body 1 and body 2
    r13 = np.sqrt((x1-x3)**2+(y1-y3)**2) # distance between body 1 and body 3
    r23 = np.sqrt((x2-x3)**2+(y2-y3)**2) # distance between body 2 and body 3

    yn[0] = y[2]    # differential equations for body 1
    yn[1] = y[3]
    yn[2] = (-m2*G/r12**3)*(x1-x2)+(-m3*G/r13**3)*(x1-x3)
    yn[3] = (-m2*G/r12**3)*(y1-y2)+(-m3*G/r13**3)*(y1-y3)

    yn[4] = y[6]    # differential equations for body 2
    yn[5] = y[7]
    yn[6] = (-m1*G/r12**3)*(x2-x1)+(-m3*G/r23**3)*(x2-x3)
    yn[7] = (-m1*G/r12**3)*(y2-y1)+(-m3*G/r23**3)*(y2-y3)

    yn[8] = y[10]   # differential equations for body 3
    yn[9] = y[11]
    yn[10] = (-m1*G/r13**3)*(x3-x1)+(-m2*G/r23**3)*(x3-x2)
    yn[11] = (-m1*G/r13**3)*(y3-y1)+(-m2*G/r23**3)*(y3-y2)

    return yn

```

```python
class Body:
    def __init__(self,mass,position,velocity=np.array([0.,0.])):
        self.mass = mass              # mass of the body
        self.position = position     # inital position vector
        self.velocity = velocity     # inital velocity vector

def initial_conditions(body1,body2,body3):
    ''' Function to write the inital conditions into an numpy array '''
    return np.array([body1.position[0],body1.position[1],
                     body1.velocity[0],body1.velocity[1],
                     body2.position[0],body2.position[1],
                     body2.velocity[0],body2.velocity[1],
                     body3.position[0],body3.position[1],
                     body3.velocity[0],body3.velocity[1]])

def min_separation(yn,xn):
    ''' Function to compute the separation between two bodies
        and store this data in a file '''
    separation = np.zeros((len(xn),4))
    separation[:,0] = xn     # time column
    # compute the distance between two bodies
    separation[:,1] = np.sqrt((yn[:,0]-yn[:,4])**2+(yn[:,1]-yn[:,5])**2)
    separation[:,2] = np.sqrt((yn[:,0]-yn[:,8])**2+(yn[:,1]-yn[:,9])**2)
    separation[:,3] = np.sqrt((yn[:,4]-yn[:,8])**2+(yn[:,5]-yn[:,9])**2)

    # find the minimum distances between two bodies
    index_min_12 = argrelextrema(separation[:,1], np.less)[0]
    index_min_13 = argrelextrema(separation[:,2], np.less)[0]
    index_min_23 = argrelextrema(separation[:,3], np.less)[0]

    min_separation_12 = np.array([separation[:,0][index_min_12],
                                  separation[:,1][index_min_12]])
    min_separation_13 = np.array([separation[:,0][index_min_13],
                                  separation[:,2][index_min_13]])
    min_separation_23 = np.array([separation[:,0][index_min_23],
                                  separation[:,3][index_min_23]])

    # store the results into txt-files
    np.savetxt('data/separation.txt', separation, delimiter='\t')
    np.savetxt('data/min_separation_12.txt', min_separation_12, delimiter='\t')
    np.savetxt('data/min_separation_13.txt', min_separation_13, delimiter='\t')
    np.savetxt('data/min_separation_23.txt', min_separation_23, delimiter='\t')

def error_total_energy(G,body1,body2,body3,yn):
    ''' Function to compute the relative error of the total energy
        of the system compared to the initial value '''

    # compute the total kinetic energy of the system
    kin_energy = 0.5*(body1.mass*(yn[:,2]**2+yn[:,3]**2)
                     +body2.mass*(yn[:,6]**2+yn[:,7]**2)
                     +body3.mass*(yn[:,10]**2+yn[:,11]**2))

    # compute the total potential energy of the system
    r12 = np.sqrt((yn[:,0]-yn[:,4])**2+(yn[:,1]-yn[:,5])**2)
    r13 = np.sqrt((yn[:,0]-yn[:,8])**2+(yn[:,1]-yn[:,9])**2)
    r23 = np.sqrt((yn[:,4]-yn[:,8])**2+(yn[:,5]-yn[:,9])**2)
    pot_energy = -G*((body1.mass*body2.mass/r12)
                    +(body1.mass*body3.mass/r13)
                    +(body2.mass*body3.mass/r23))

```

```
    # compute the total energy of the system and the relative error
    total_energy = kin_energy+pot_energy
    relative_error = abs(total_energy-total_energy[0])/abs(total_energy[0])
    return relative_error


G = 1.  # simplify the system by setting the gravitational constant to G=1
# create the three bodies with their initial conditions:
# Meissel-Burrau problem or Pythagorean problem
body1 = Body(3., np.array([1.,3.]))
body2 = Body(4., np.array([-2.,-1.]))
body3 = Body(5., np.array([1.,-1.]))

# numerical simulation of the gravitational three-body problem
# using the Runge-Kutta-4 integrator
yn, xn = rk4(initial_conditions(body1,body2,body3),0,three_body_problem,4e-5,int(5e5),
                  {'G':G, 'm1':body1.mass, 'm2':body2.mass, 'm3':body3.mass})


min_separation(yn,xn) # compute the minimum separation
separation = np.loadtxt('data/separation.txt')  # load distance data from files
min_sep_12 = np.loadtxt('data/min_separation_12.txt')
min_sep_13 = np.loadtxt('data/min_separation_13.txt')
min_sep_23 = np.loadtxt('data/min_separation_23.txt')

# compute the total energy and the relative error of the total energy
relative_error = error_total_energy(G,body1,body2,body3,yn)


# plot the trajectories of the three bodies in the orbital plane
fig, ax = plt.subplots()
ax.set_title('Numerical Simulation of the Gravitational Three-Body Problem\n'+
             'trajectories of the three bodies in the orbital plane')
ax.set_xlabel(r'$x$-coordinates'); ax.set_ylabel(r'$y$-coordinates')
ax.plot(yn[:,0], yn[:,1], 'r.', markersize=1, label='Body 1')
ax.plot(yn[:,4], yn[:,5], 'g.', markersize=1, label='Body 2')
ax.plot(yn[:,8], yn[:,9], 'b.', markersize=1, label='Body 3')
ax.set_xlim((-3.5,3.5)); ax.set_ylim((-3,4))
ax.grid(); ax.legend(loc='best', markerscale=8)
fig.savefig('figures/Meissel-Burrau_Trajectories.png', format='png')

# plot the time evolution of the distance between two bodies
fig, ax = plt.subplots()
ax.set_title('Numerical Simulation of the Gravitational Three-Body Problem\n'+
             'time evolution of the distance between two bodies')
ax.set_xlabel('time'); ax.set_ylabel('distance')
ax.plot(separation[:,0], separation[:,1], 'r.', markersize=1, label='Bodies 1 and 2')
ax.plot(separation[:,0], separation[:,2], 'g.', markersize=1, label='Bodies 1 and 3')
ax.plot(separation[:,0], separation[:,3], 'b.', markersize=1, label='Bodies 2 and 3')
ax.grid(); ax.legend(loc='best', markerscale=8); ax.set_yscale('log')
fig.savefig('figures/Meissel-Burrau_Distances.png', format='png')

# plot the time evolution of the minimum separation between two bodies
fig, ax = plt.subplots()
ax.set_title('Numerical Simulation of the Gravitational Three-Body Problem\n'+
```

```
              'time evolution of the minimum separation between two bodies')
ax.set_xlabel('time'); ax.set_ylabel('distance')
ax.plot(min_sep_12[0], min_sep_12[1], 'rx', label='Bodies 1 and 2')
ax.plot(min_sep_13[0], min_sep_13[1], 'gx', label='Bodies 1 and 3')
ax.plot(min_sep_23[0], min_sep_23[1], 'bx', label='Bodies 2 and 3')
ax.grid(); ax.legend(loc='best'); ax.set_yscale('log')
fig.savefig('figures/Meissel-Burrau_Minimum-Separation.png', format='png')

# plot the time evolution of the relative error of the total energy
fig, ax = plt.subplots()
ax.set_title('Numerical Simulation of the Gravitational Three-Body Problem\n'+
              'time evolution of the relative error of the total energy')
ax.set_xlabel('time'); ax.set_ylabel('relative error of the total energy')
ax.plot(xn, relative_error, 'b.', markersize=1); ax.grid(); ax.set_yscale('log')
fig.savefig('figures/Meissel-Burrau_Realtive-Error.png', format='png')

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').