Introduction to Computational Physics - Exercise 2

Simon Groß-Bölting, Lorenz Vogel, Sebastian Willenberg

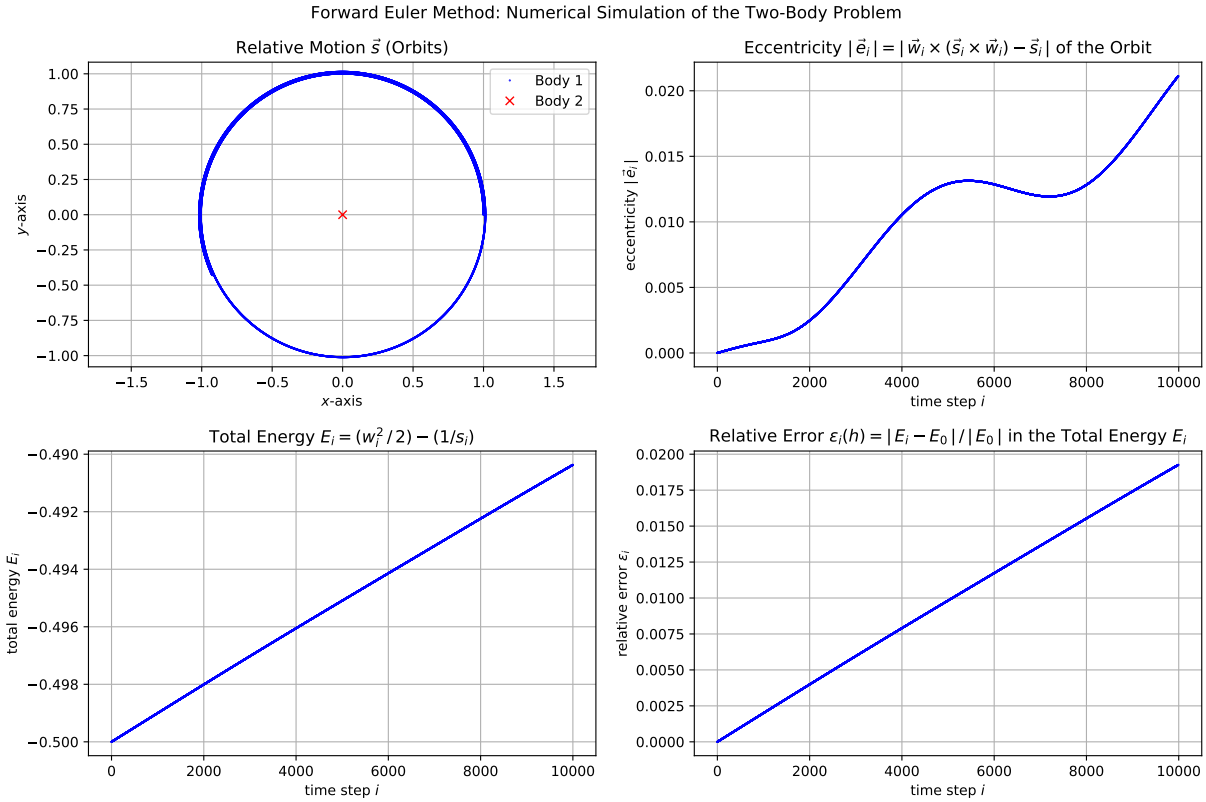
8. April 2020

In the first exercise form the worksheet we worked on the following code:

```python
import numpy as np
import matplotlib.pyplot as plt

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

def forward_euler(body1,body2,G,dt,N):
    ''' This function computes the relative motion of two point-like bodies
        under their mutual gravitational influence using a
        forward Euler integration procedure
        Input:  two bodies of the class "Body", gravitational constant G,
                time steps dt and number of time steps N
        Output: location vector s and velocity vector w '''

    # compute the total mass of the two bodies and set the
    # characteristic length scale as the inital seperation
    M = body1.mass+body2.mass
    R0 = np.linalg.norm(body1.position-body2.position)
    h = dt/np.sqrt(R0**3/(G*M)) # compute the dimensionless step size

    s = np.zeros((int(N),3))    # create array for the location vectors
    s[0] = (body1.position-body2.position)/R0

    w = np.zeros((int(N),3))    # create array for the velocity vectors
    w[0] = (body1.velocity-body2.velocity)/np.sqrt(G*M/R0)

    for i in range(1,int(N)):   # compute the relative motion
        s[i] = s[i-1]+(w[i-1]*dt)
        w[i] = w[i-1]-(s[i-1]/np.linalg.norm(s[i-1])**3.)*dt
    return (s,w)

def get_acceleration(body1,body2,G,position):
    return -(body1.mass+body2.mass)*G*position/np.linalg.norm(position)**3

def leapfrog(body1,body2,G,dt,N):
    ''' This function computes the relative motion of two point-like bodies
        under their mutual gravitational influence using a Leapfrog scheme
        Input:  two bodies of the class "Body", gravitational constant G,
                time steps dt and number of time steps N
        Output: location vector s and velocity vector w '''

    # compute the total mass of the two bodies and set the
    # characteristic length scale as the inital seperation
    M = body1.mass+body2.mass
    R0 = np.linalg.norm(body1.position-body2.position)
    h = dt/np.sqrt(R0**3/(G*M)) # compute the dimensionless step size
```

```
50
51      s = np.zeros((int(N),3))    # create array for the location vectors
52      s[0] = (body1.position-body2.position)/R0
53
54      w = np.zeros((int(N),3))    # create array for the velocity vectors
55      w[0] = (body1.velocity-body2.velocity)/np.sqrt(G*M/R0)
56
57      w[0] -= (s[0]/np.linalg.norm(s[0])**3)*h/2
58      for i in range(1,int(N)):
59          w[i] = w[i-1]+get_acceleration(body1, body2, G, s[i-1])*h
60          s[i] = s[i-1]+w[i]*h
61      return (s,w)
62
63
64  def total_energy(s,w):
65      ''' Function to compute the total energy of the system from the
66          dimensionless vectors s (location) and w (velocity) '''
67      energy = np.zeros(np.shape(s)[0])
68      for i in range(0,np.shape(s)[0]):
69          energy[i] = (0.5*np.linalg.norm(w[i])**2)-(1/np.linalg.norm(s[i]))
70      return energy
71
72  def relative_error(energy):
73      ''' Function to compute the relative error in the total energy at
74          step i compared to the initial value 0 '''
75      rel_error = np.zeros(np.shape(s)[0])
76      for i in range(0,np.shape(s)[0]):
77          rel_error[i] = abs(energy[i]-energy[0])/abs(energy[0])
78      return rel_error
79
80  def angular_momentum(s,w):
81      ''' Function to compute the angular momentum'''
82      angular_momentum = np.zeros((np.shape(s)[0],3))
83      for i in range(0,np.shape(s)[0]):
84          angular_momentum[i] = np.cross(s[i],w[i])
85      return angular_momentum
86
87  def eccentricity(s,w):
88      ''' Function to compute the eccentricity
89          from the Laplace-Runge-Lenz vector '''
90      LRL = np.zeros((np.shape(s)[0],3))
91      for i in range(0,np.shape(s)[0]):
92          LRL[i] = np.cross(w[i], np.cross(s[i],w[i]))-s[i]
93      return np.linalg.norm(LRL, axis=1)
94
95  # set the initial properties of the two-body system
96  G = 1.  # set the gravitational constant
97  body1 = Body(1., np.array([1.,0.,0.]))
98  body2 = Body(1., np.array([0.,0.,0.]))
99  dt = 1e-3;  N=1e4   # set step length and number of steps
100
101 # compute the initial velocity for a circular orbit
102 v0 = np.sqrt(G*(body1.mass+body2.mass)/np.linalg.norm(body1.position-body2.
        position))
103 body1.velocity = np.array([0.,v0,0.])
104
105
106 ## Numerical Simulation of the Two-Body Problem using Forward Euler Method
107 fig, axs = plt.subplots(2,2,figsize=(12,8), constrained_layout=True)
108 fig.suptitle(r'Forward Euler Method: Numerical Simulation of the Two-Body
        Problem')
109
110 s, w = forward_euler(body1, body2, G, dt, N)
```

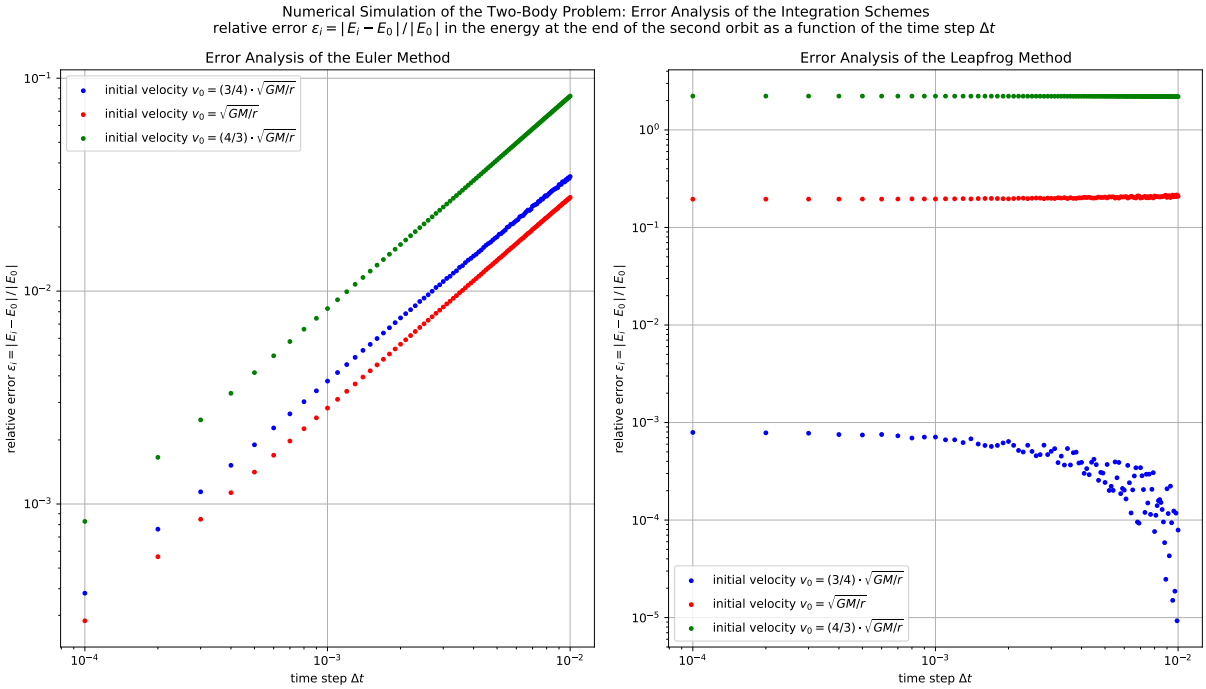
```
energy = total_energy(s,w)                    # compute the total energy
rel_error = relative_error(energy)            # compute the relative error
angular_momentum = angular_momentum(s,w)      # compute the angular momentum
eccentricity = eccentricity(s,w)              # compute the eccentricity

axs[0,0].plot(s[:,0], s[:,1], 'b.', markersize=1, label='Body 1')
axs[0,0].plot([0], [0], 'rx', label='Body 2')
axs[0,0].set_title(r'Relative Motion $\vec{s}$ (Orbits)')
axs[0,0].set_xlabel(r'$x$-axis'); axs[0,0].set_ylabel(r'$y$-axis')
axs[0,0].axis('equal'); axs[0,0].legend(loc='best')

axs[0,1].plot(range(0,len(eccentricity)), eccentricity, 'b.', markersize=1)
axs[0,1].set_title(r'Eccentricity $\vert\,\vec{e}_i\,\vert=\vert\,\vec{w}_i\
    times'
                   +r'(\vec{s}_i\times\vec{w}_i)-\vec{s}_i\,\vert$ of the Orbit'
    )
axs[0,1].set_xlabel(r'time step $i$')
axs[0,1].set_ylabel(r'eccentricity $\vert\,\vec{e}_i\,\vert$')

axs[1,0].plot(range(0,len(energy)), energy, 'b.', markersize=1)
axs[1,0].set_title(r'Total Energy $E_i=(w_i^2\,/\,2)-(1/s_i)$')
axs[1,0].set_xlabel(r'time step $i$'); axs[1,0].set_ylabel(r'total energy $E_i$
    ')

axs[1,1].plot(range(0,len(rel_error)), rel_error, 'b.', markersize=1)
axs[1,1].set_title(r'Relative Error $\epsilon_i(h)=\vert\,E_i-E_0\,\vert\,/\,\
    vert\,E_0\,\vert$'
                   +r' in the Total Energy $E_i$')
axs[1,1].set_xlabel(r'time step $i$'); axs[1,1].set_ylabel(r'relative error $\
    epsilon_i$')

for ax in fig.get_axes():
    ax.grid()
fig.savefig('figures/Two-Body-Problem_Euler-Method.pdf', format='pdf', dpi=100)
```



2 Error Analysis of Euler Scheme

(a) After varying the inital velocity three times we can see that the duble logarithmic plot of the function is always a straight line with the same slope but different starting points. Therefore all the lines are parallel. This differs quite a lot from what we would expect. The law of conservation of energy predicts, that the energy in the system should remain constant. But we actually see that the energy in the system increases. This also becomes apparent in the plot with the relative motion. We can see that the path of the second body deviates slightly from the circular path it is supposed to take when the initial velocity is set to $v_0 = 2\sqrt{\frac{GM}{R_0}}$. That means that the distance between both bodies increases with each orbit.

```
## Error Analysis of the Euler and the Leapfrog Scheme
v0_range = [(3/4)*v0, v0, (4/3)*v0]             # set the initial velocities
dt_range = np.linspace(1e-4, 1e-2, int(1e2))    # set different time steps

# compute the relative error in the energy for Euler and Leapfrog
rel_error_euler = np.zeros((len(v0_range),len(dt_range)))
rel_error_leapfrog = np.zeros((len(v0_range),len(dt_range)))

for i in range(0,len(v0_range)):
    for j in range(0,len(dt_range)):
        N = int(2*np.sqrt(1.**3/(1.*2.))/dt_range[j])
        body1.velocity = np.array([0.,v0_range[i],0.])

        s, w = forward_euler(body1, body2, G, dt_range[j], N)
        rel_error_euler[i,j] = relative_error(total_energy(s,w))[N-1]

        s, w = leapfrog(body1, body2, G, dt_range[j], N)
        rel_error_leapfrog[i,j] = relative_error(total_energy(s,w))[N-1]

# plot the relative error in the energy for Euler and Leapfrog
fig, axs = plt.subplots(1,2,figsize=(14,8), constrained_layout=True)
fig.suptitle('Numerical Simulation of the Two-Body Problem: Error Analysis of the Integration Schemes\n'
             +r'relative error $\epsilon_i=\vert\,E_i-E_0\,\vert\,/\,\vert\,E_0\,\vert$ in the energy'
             +r' at the end of the second orbit as a function of the time step $\Delta t$')
label = [r'initial velocity $v_0=(3/4)\cdot\sqrt{GM/r}$',
         r'initial velocity $v_0=\sqrt{GM/r}$',
         r'initial velocity $v_0=(4/3)\cdot\sqrt{GM/r}$']
color = ['b.','r.','g.']
axs[0].set_title('Error Analysis of the Euler Method')
axs[1].set_title('Error Analysis of the Leapfrog Method')

for i in range(0,3):
    axs[0].plot(dt_range, rel_error_euler[i,:], color[i], label=label[i])
    axs[1].plot(dt_range, rel_error_leapfrog[i,:], color[i], label=label[i])

for ax in fig.get_axes():
    ax.set_xlabel(r'time step $\Delta t$')
    ax.set_ylabel(r'relative error $\epsilon_i=\vert\,E_i-E_0\,\vert\,/\,\vert\,E_0\,\vert$')
    ax.grid(); ax.legend(loc='best')
    ax.set_xscale('log'); ax.set_yscale('log')
fig.savefig('figures/Two-Body-Problem_Error-Analysis.pdf', format='pdf', dpi=100)
plt.show(); plt.clf(); plt.close()
```



(b) After implementing the Leapfrog scheme we can see, that the energy remains constant for initial velocities that are smaller or equal to $v_0 = 2\sqrt{\frac{GM}{R_0}}$. This means that the conservation of enrgy applies here. If we plot the motion in this case we can see that the circular remains equidistant to the center. But if we look at initial velocities that are greater than $v_0 = 2\sqrt{\frac{GM}{R_0}}$ we can see that the energy of the system doesn't remain constant. The energy actually decreases in this case.