Computational physics

Jona Ackerschott, Julian Mayr

Problem set 2

Problem 1

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

# General implementation of 4th order runge kutta algorithm
# ODE: u'(t) = f(u(t), t) where u(t0) = u0
def rk4(f, t0, u0, dt, N):
  t = np.zeros(N)
  u = np.zeros(N)

  t[0] = t0
  u[0] = u0
  for i in range(1, N):
    k1 = f(t[i - 1], u[i - 1])
    k2 = f(t[i - 1] + dt / 2, u[i - 1] + k1 * dt / 2)
    k3 = f(t[i - 1] + dt / 2, u[i - 1] + k2 * dt / 2)
    k4 = f(t[i - 1] + dt, u[i - 1] + k3 * dt)
    ud = (k1 + 2 * k2 + 2 * k3 + k4) / 6

    t[i] = t[i - 1] + dt
    u[i] = u[i - 1] + ud * dt
  return t, u

# RHS of the ODE
def f(u, r):
  return -r * u

# Analytical solution
def u(t, r):
  return exp(-r * t)

# Arrays for the plot of the analytical solution
t_a = np.linspace(0, 10, 1000)
u_a = u(t_a, 1)

# Plot numerical and analytical solutions normal and logarithmic
fig1, ax1 = plt.subplots()
fig2, ax2 = plt.subplots()
```

```
ax1.semilogy(t_a, u_a)
ax2.plot(t_a, u_a)
for N in [5, 6, 7, 10, 12]:
  t, u = rk4(lambda t, u: f(u, 1), 0, 1, 10 / (N - 1), N)
  ax1.semilogy(t, u)
  ax2.plot(t, u)

fig1.savefig('sols_ja/pset2/figures/fig1_1.png', papertype='a4',
  orientation='landscape', bbox_inches='tight', format='png')
fig2.savefig('sols_ja/pset2/figures/fig1_2.png', papertype='a4',
  orientation='landscape', bbox_inches='tight', format='png')

plt.show()
```

Source code 1: problem1.py

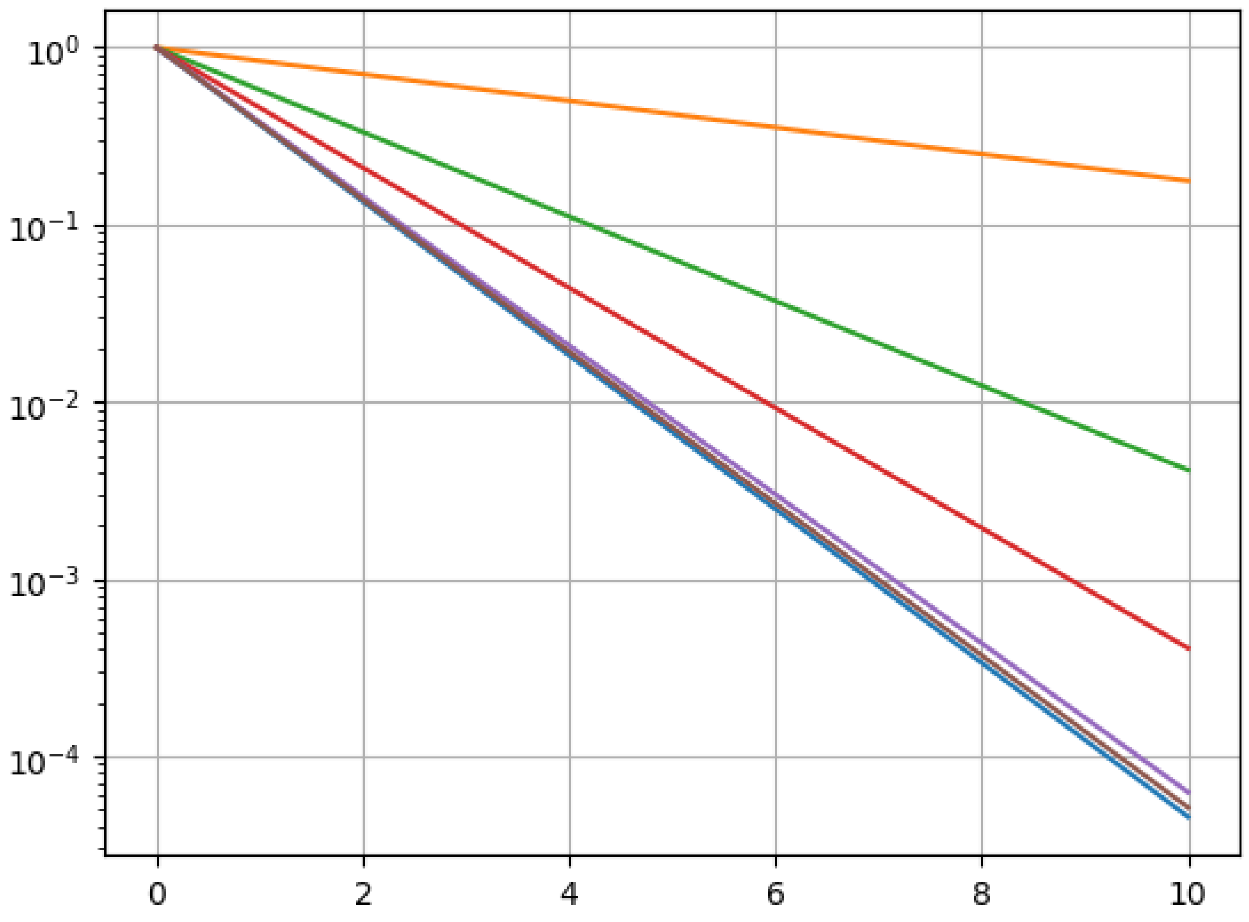

Figure 1: Semi logarithmic plot of the analytical (blue) and the numerical solutions. For the numerical solutions (orange, green, red, purple and brown), the step sizes $2.0, 1.7, 1.4, 1.0$ and 0.8 where used.

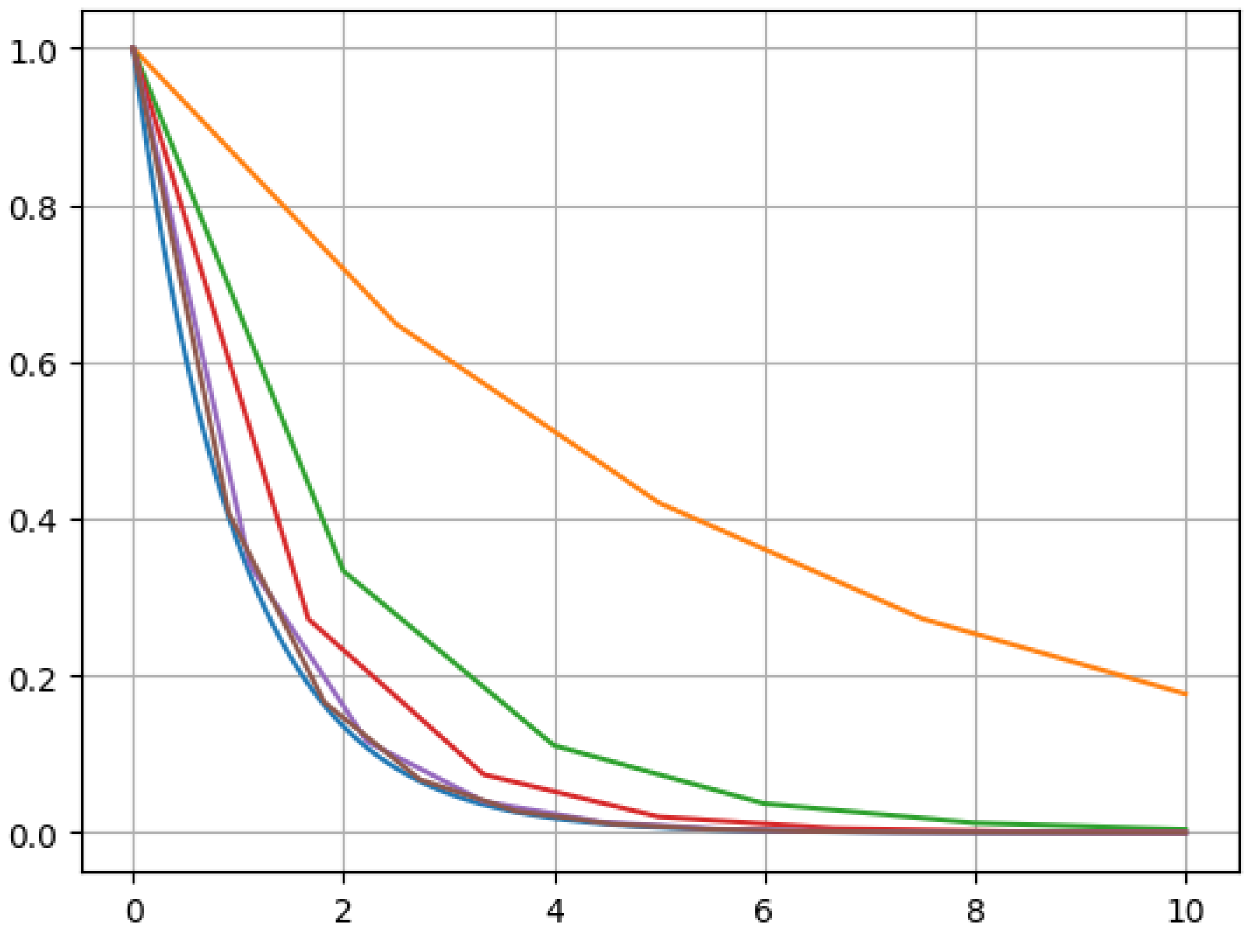

Figure 2: Normal plot of the analytical (blue) and the numerical solutions from figure 1.

From figure 1 one can see, that the slopes of the numerical solutions are converging pretty fast to the slope of the analytical solution and that with relatively large step sizes. The purple solution is already, at least visually, a decent approximation of the blue curve (at the points where it is not interpolated) with a step size of 1.0. So one can see that the 4th order runge-kutta method seems to produce very good results in practice.

Problem 2

```
import matplotlib.pyplot as plt
import numpy as np
from numpy import sqrt, dot
from numpy.linalg import norm

# Not important for the problem, but rather useful for long computation times
def printProgressBar (iteration, total, prefix = '', suffix = '',
  decimals = 1, length = 100, fill = ''):
    perc = 100 * (iteration / float(total))
    percStr = ("{0:." + str(decimals) + "f}").format(perc)
    filledLength = int(length * iteration // total)
    bar = fill * filledLength + '-' * (length - filledLength)
    print('%s |%s| %s%% %s\r' % (prefix, bar, percStr, suffix), end = '')
    # Print New Line on Complete
    if iteration == total:
        print()

# Differential equation for the second derivative
def xdd(x, m):
  return np.array([
    -m[1] * (x[0] - x[1]) / norm(x[0] - x[1])**3
      - m[2] * (x[0] - x[2]) / norm(x[0] - x[2])**3,
    -m[2] * (x[1] - x[2]) / norm(x[1] - x[2])**3
      - m[0] * (x[1] - x[0]) / norm(x[1] - x[0])**3,
    -m[0] * (x[2] - x[0]) / norm(x[2] - x[0])**3
      - m[1] * (x[2] - x[1]) / norm(x[2] - x[1])**3
  ])

# Returns the trajectories (times, points and tangent vectors)
# for 3 bodies interacting through gravity. This function works
# with 3xNx2 arrays (N 2-dim. vectors for 3 bodies)
def body3traj(x0, xd0, m, dt, N):
  t = np.zeros(N)
  x, xd = np.zeros((3, N, 2)), np.zeros((3, N, 2))
  x[:, 0] = x0
  xd[:, 0] = xd0
  for i in range(1, N):
    printProgressBar(i, N - 1, suffix='Complete', prefix='Progress')

    t[i] = t[i - 1] + dt

    # Runge kutta for the two first order equations for each body
    k1 = xdd(x[:, i - 1], m)
    l1 = xd[:, i - 1]
    k2 = xdd(x[:, i - 1] + l1 * dt / 2, m)
    l2 = xd[:, i - 1] + k2 * dt / 2
    k3 = xdd(x[:, i - 1] + l2 * dt / 2, m)
    l3 = xd[:, i - 1] + k3 * dt / 2
    k4 = xdd(x[:, i - 1] + l3 * dt, m)
    l4 = xd[:, i - 1] + k4 * dt

    x[:, i] = x[:, i - 1] + (l1 + 2 * l2 + 2 * l3 + l4) * dt / 6
    xd[:, i] = xd[:, i - 1] + (k1 + 2 * k2 + 2 * k3 + k4) * dt / 6
  return t, x, xd

# Returns the mutual distances from trajectories as 3xNx2 array
def body3mutDist(x):
  x_next = np.roll(x, -1, axis=0)
  return norm(x_next - x, axis=2)

def body3minDistTimes(r, t, outPath=None):
  tMinDist = np.array([])
  inds = np.array([], dtype=int)
  for i in range(3):
    for j in range(1, len(t) - 1):
      if r[i, j - 1] > r[i, j] and r[i, j] < r[i, j + 1]:
        tMinDist = np.append(tMinDist, t[j])
        inds = np.append(inds, i)

  if outPath != None:
    with open(outPath, 'w') as out:
      for i in range(len(inds)):
        print(inds[i], tMinDist[i], sep='  ', file=out)
  return tMinDist

# Returns the total energy from trajectories as 3xNx2 array and masses
def body3totErg(x, xd, m):
  T, V = np.zeros(np.size(x, axis=1)), np.zeros(np.size(x, axis=1))
  for i in range(3):
    for j in range(len(xd[i])):
      T[j] += 0.5 * m[i] * dot(xd[i, j], xd[i, j])
    V += -m[i] * m[i - 2] / norm(x[i - 2] - x[i], axis=1)
    - m[i] * m[i - 1] / norm(x[i - 1] - x[i], axis=1)
  return T + V
```

Source code 2: problem2.py

(a)
```
import numpy as np
import matplotlib.pyplot as plt
import random as rng
from problem2 import body3traj

# Parameters and initial conditions
m = np.array([1, 1, 1])
x0 = np.array([
  [-0.97000436,  0.24308753],
  [ 0.97000436, -0.24308753],
  [ 0.00000000,  0.00000000]
])
xd0 = np.array([
  [-0.46620368, -0.43236573],
  [-0.46620368, -0.43236573],
  [ 0.93240737,  0.86473146]
])
dt = 0.01
N = 1000

# Compute the trajectories
t, x, xd = body3traj(x0, xd0, m, dt, N)

# Plot trajectories
fig, axes = plt.subplots(nrows=3, ncols=1)
for i in range(len(axes)):
  axes[i].axis('equal')
```

```
    axes[i].plot(x[i, :, 0], x[i, :, 1], color=('C' + str(i)))

fig.savefig('sols_ja/pset2/figures/fig2a_1.png', papertype='a4',
    orientation='landscape', bbox_inches='tight', format='png')

plt.show()
```

Source code 3: problem2a.py

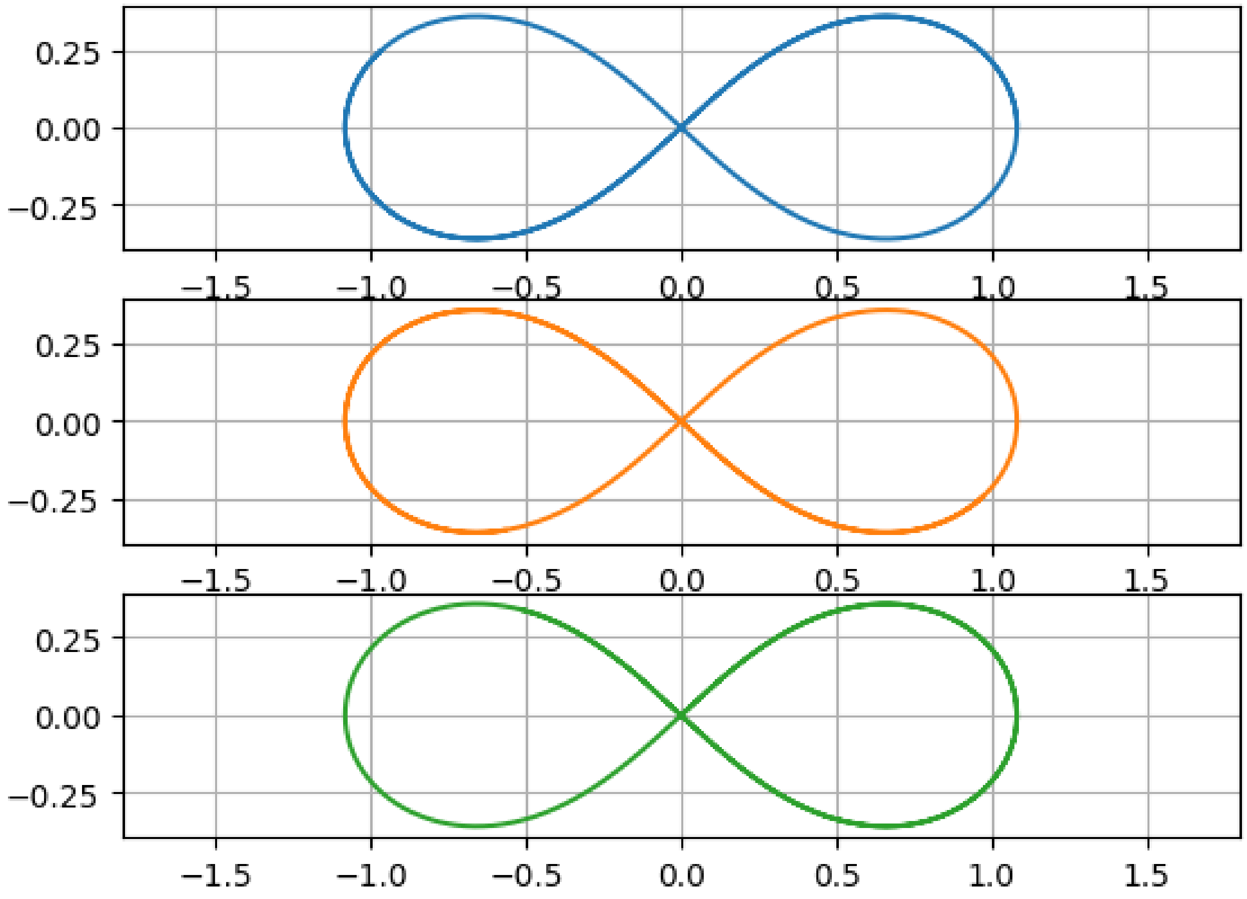

Figure 3: Trajectory plot for the 3 bodys, with the given initial conditions. The 3 Graphs lie perfectly on top of each other, so that they are drawn separately.

(b)
```
import numpy as np
import matplotlib.pyplot as plt
import random as rng
from numpy import abs, log10
from problem2 import body3traj, body3mutDist, body3minDistTimes, body3totErg

# Parameters and initial conditions
m = np.array([3.0, 4.0, 5.0])
x0 = np.array([
    [0.0, 0.0],
    [-4.0, 0.0],
    [-4.0, 3.0]
])
xd0 = np.array([
  [0.0, 0.0],
  [0.0, 0.0],
  [0.0, 0.0]
])

# Place the origin at the center of mass
X = np.sum([m[i] * x0[i] for i in range(3)], axis=0) / np.sum(m)
x0 -= X

for N in [100, 10000, 100000, 1000000]:
  dt = 10.0 / N
  n = int(log10(N)) - 1
  minDistTimesPath = 'sols_ja/pset2/out/minDistTimes_' + str(n) + '.txt'

  # Compute the trajectories, mutual distances and total energy
  t, x, xd = body3traj(x0, xd0, m, dt, N)
  r = body3mutDist(x)
  tMinDist = body3minDistTimes(r, t, outPath=minDistTimesPath)
  E = body3totErg(x, xd, m)

  # Plot trajectories
  fig1, ax1 = plt.subplots()
  plt.axis('equal')
  plt.axis([-10, 10, -10, 10])
  for i in range(3):
    plt.plot(x[i, :, 0], x[i, :, 1])

  fig2, ax2 = plt.subplots()
  for i in range(3):
    plt.semilogy(t, r[i])

  fig3, ax3 = plt.subplots()
  plt.semilogy(t, abs(1 - E / E[0]))

  fig1.savefig('sols_ja/pset2/figures/fig2b_' + str(3 * n - 2) + '.png',
    papertype='a4', orientation='landscape', bbox_inches='tight', format='png')
  fig2.savefig('sols_ja/pset2/figures/fig2b_' + str(3 * n - 1) + '.png',
    papertype='a4', orientation='landscape', bbox_inches='tight', format='png')
  fig3.savefig('sols_ja/pset2/figures/fig2b_' + str(3 * n) + '.png',
    papertype='a4', orientation='landscape', bbox_inches='tight', format='png')

plt.show()
```

Source code 4: problem2b.py

As one can see from the figures 4-7, the behaviour of this system is chaotic, which means in this case, that the trajectories are very different even for the 4th order runge kutta method with very small step sizes. The first „good“ result one gets in this problem, is, in terms of the energy uncertainty, with a step size of $\Delta t = 0.00001$, which yields a first maximal energy uncertainty which is strictly less than one (cf. figure 7).
If one takes a look at the times of the first 5 encounters of the 3 bodys, one gets the

following results from data 1-4 for $\Delta t = 0.1, 0.001, 0.0001$ and 0.00001:

n	0.1	0.001	0.0001	0.00001
1	1.80000	1.82800	1.82760	1.82764
2	1.80000	1.88000	1.88030	1.88033
3	1.90000	1.88300	2.99670	3.11570
4		1.88400	3.00800	3.20262
5		1.89800	3.02040	3.97313

Table 1: Time points of the first 5 encounters (minimal distance) of the 3 bodys, for different values of Δt

So as one can see, the first decimal place of the time points seems to be accurate (maybe except for the last value) at $\Delta t = 0.0001$. So with $\Delta t = 0.00001$ one should get a pretty good guess for these values, especially the first few. But to determine this values accurately to a few decimal places one probably needs a step size which is some orders of magnitude below that. Unfortunately, to do this with the current code in hand is rather impractical, because the computation time for $\Delta t = 0.00001$ is already at 15 Minutes. So if one would take theses further computations seriously, he should think about using C/C++. However, the use of C/C++ would certainly lead to another problem regarding the dimensionality of the problem. This is not a problem in python, because the numpy package can handle mutlidimensional arrays pretty well, which would certainly be a nightmare in C/C++, if one thinks about memory handling.

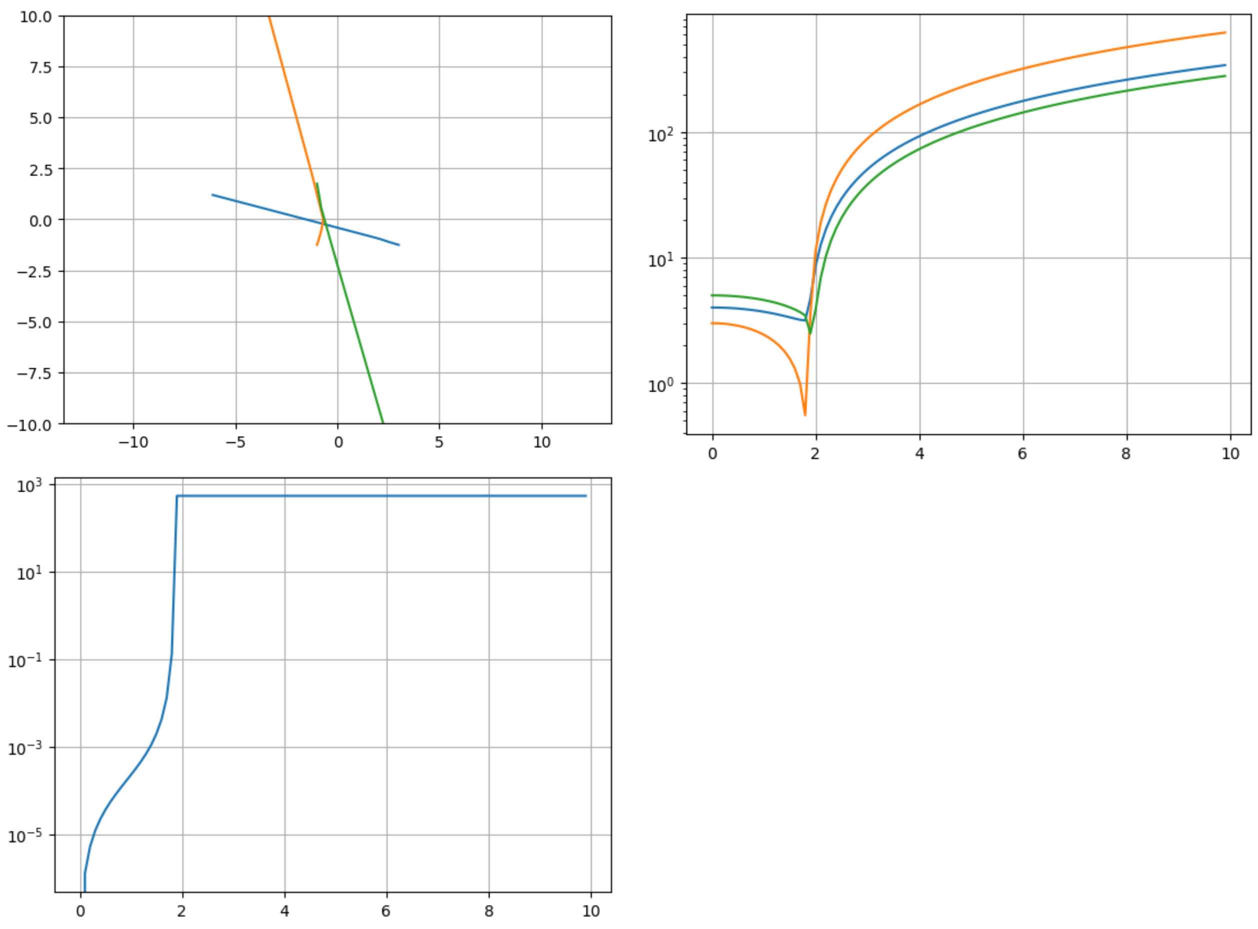

Figure 4: Plots of the trajectories (body 1: blue, body 2: orange, body 3: green), mutual distances (body 1-2: blue, body 2-3: orange, body 3-1: green) and the error $\left|1 - \frac{E}{E_0}\right|$ of the total Energy E, the last two as a function of time, for $\Delta t = 0.1$.

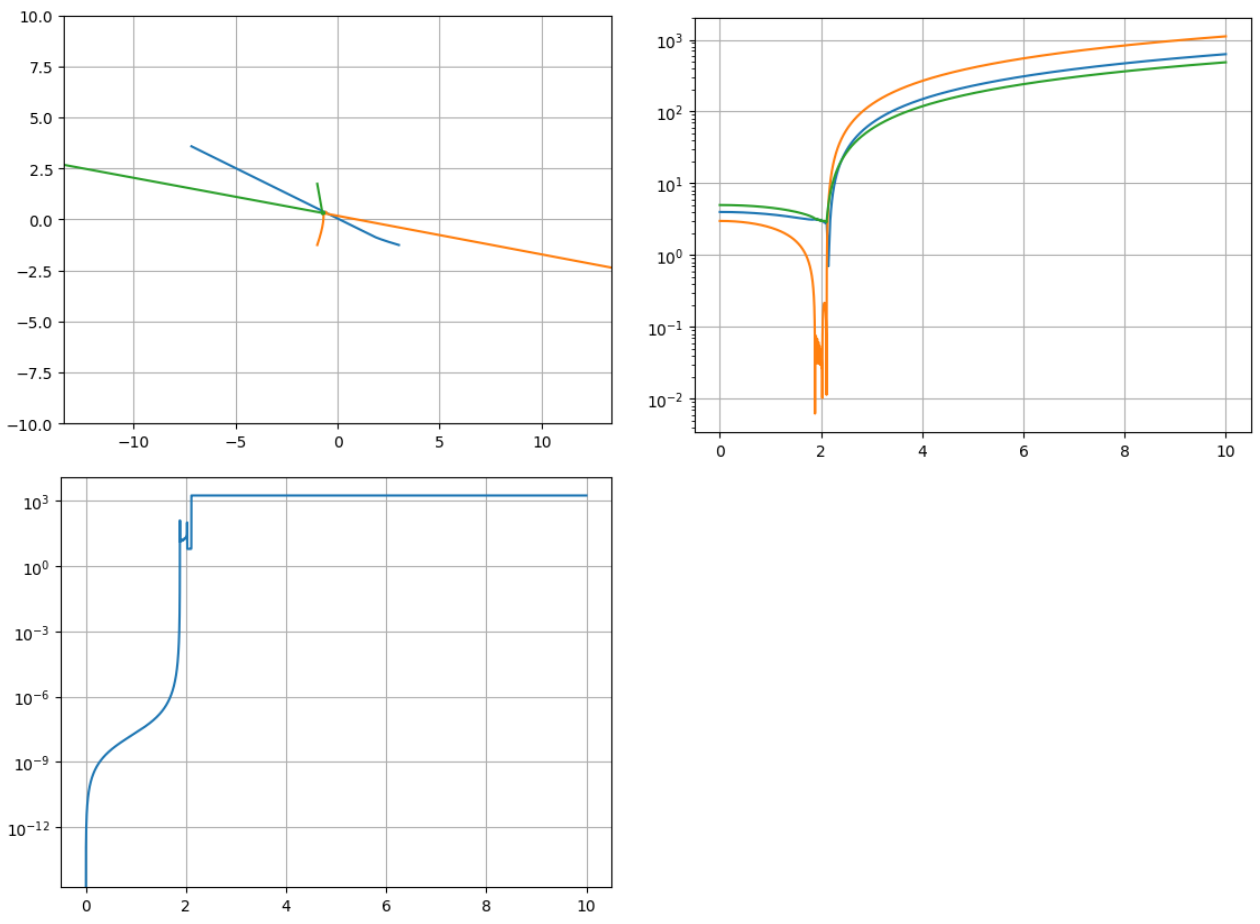

Figure 5: Plots of the trajectories (body 1: blue, body 2: orange, body 3: green), mutual distances (body 1-2: blue, body 2-3: orange, body 3-1: green) and the error $\left|1 - \frac{E}{E_0}\right|$ of the total Energy E, the last two as a function of time, for $\Delta t = 0.001$.

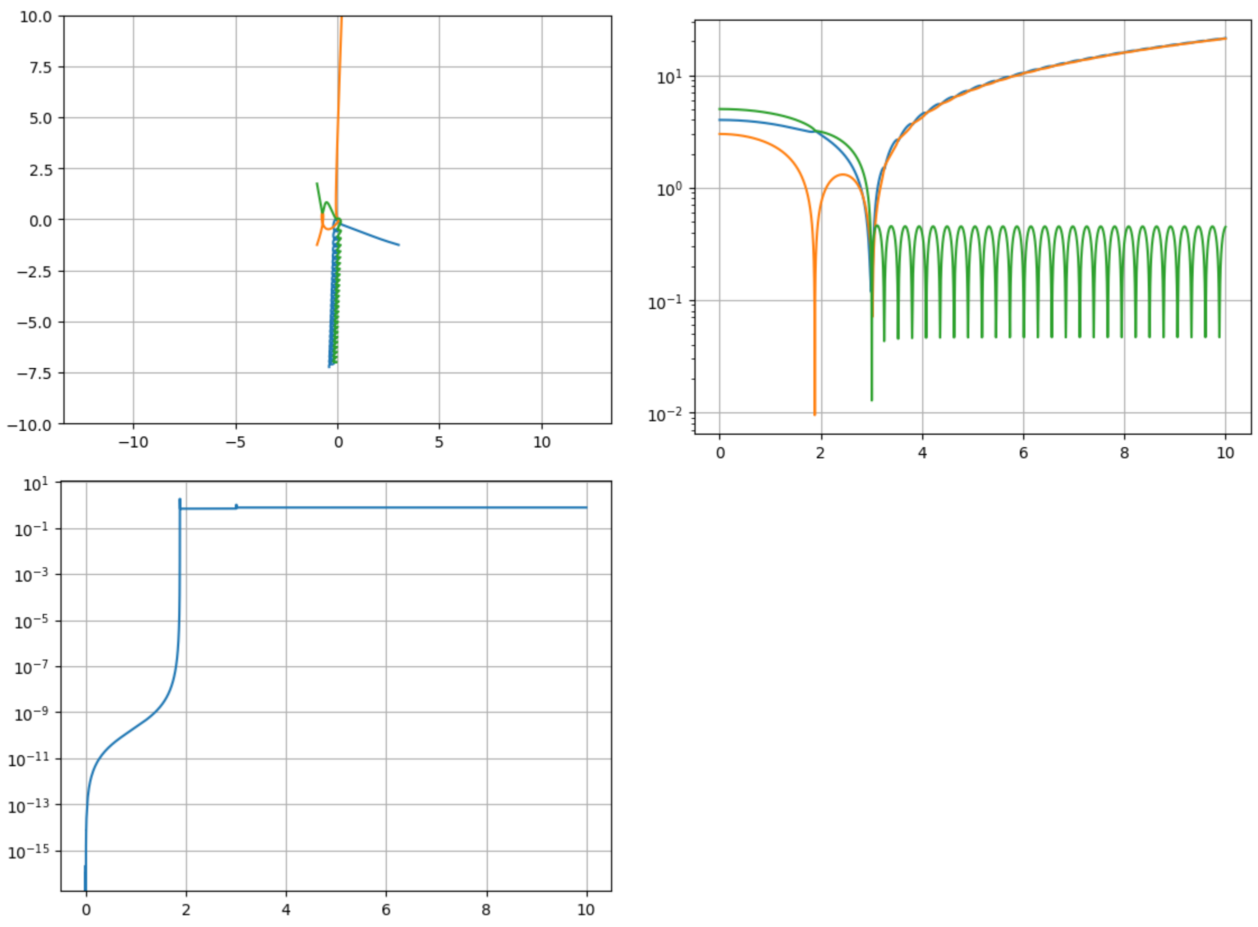

Figure 6: Plots of the trajectories (body 1: blue, body 2: orange, body 3: green), mutual distances (body 1-2: blue, body 2-3: orange, body 3-1: green) and the error $\left|1 - \frac{E}{E_0}\right|$ of the total Energy E, the last two as a function of time, for $\Delta t = 0.0001$

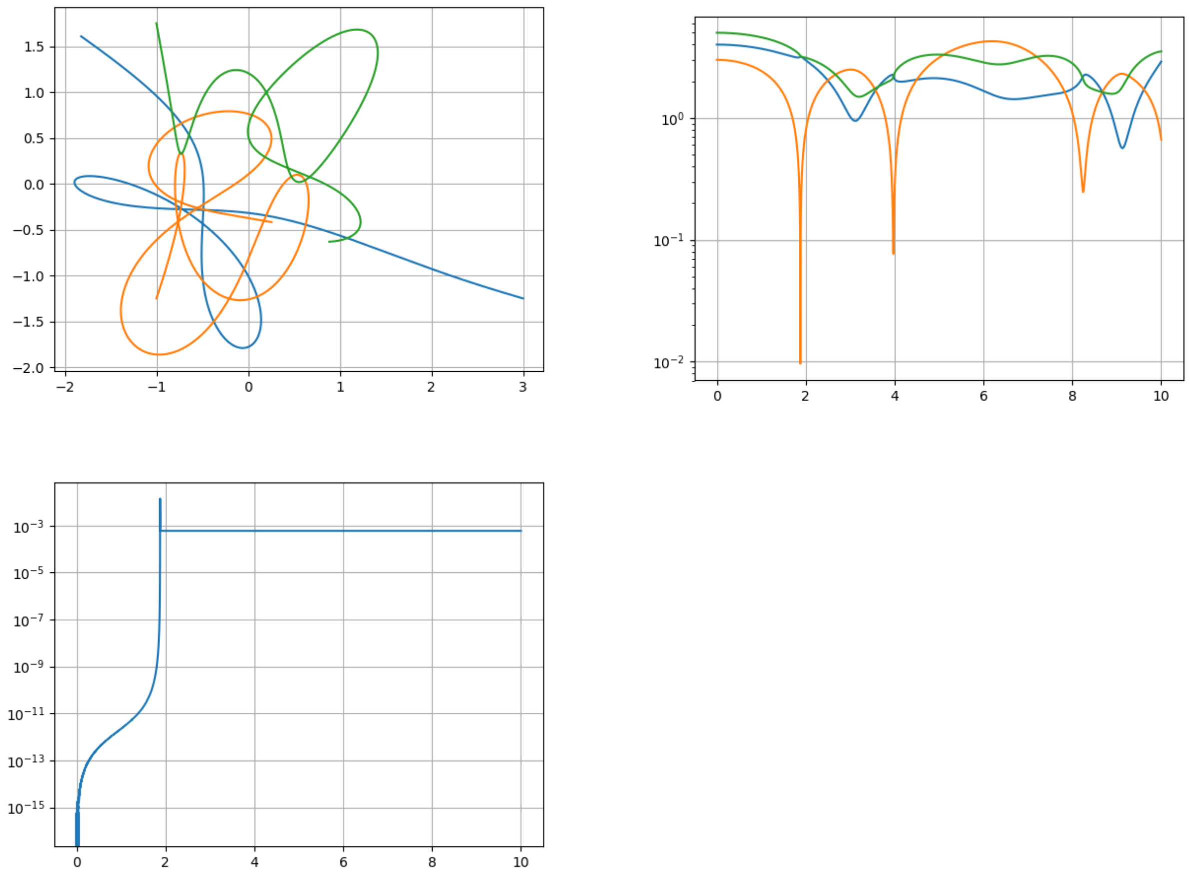

Figure 7: Plots of the trajectories (body 1: blue, body 2: orange, body 3: green), mutual distances (body 1-2: blue, body 2-3: orange, body 3-1: green) and the error $\left|1 - \frac{E}{E_0}\right|$ of the total Energy E, the last two as a function of time, for $\Delta t = 0.00001$

```
0  1.8000000000000005
1  1.8000000000000005
2  1.9000000000000006
```

Data 1: Time points with minimal mutual distances of the bodys, for $\Delta t = 0.1$. The body is given by the index.

```
0  1.8279999999999095
0  1.8979999999999018
0  1.9219999999998991
0  1.9439999999998967
0  1.9639999999998945
0  1.9819999999998925
0  1.9979999999998908
0  2.0119999999998894
0  2.0219999999998883
0  2.0909999999998807
0  2.142999999999875
1  1.8799999999999037
1  1.8829999999999034
1  1.9109999999999003
1  1.9359999999998976
1  1.9579999999998952
1  1.977999999999893
1  1.9969999999998909
1  2.0129999999998893
1  2.022999999999888
1  2.107999999999879
2  1.8839999999999033
2  1.9099999999999004
2  1.9339999999998978
2  1.9559999999998954
2  1.9749999999998933
2  1.9919999999998914
2  2.00599999999989
2  2.016999999999889
2  2.024999999999888
2  2.107999999999879
```

Data 2: Time points with minimal mutual distances of the bodys, for $\Delta t = 0.001$. The body is given by the index.

```
0  1.8275999999998152
0  2.9967000000018995
0  3.0208000000019504
0  3.2554000000024454
0  3.5318000000030287
0  3.808200000003612
0  4.08440000000382
0  4.360700000003176
0  4.636900000002532
0  4.913100000001888
```

```
0  5.189300000001245
0  5.465500000000601
0  5.741599999999957
0  6.017699999999314
0  6.29389999999867
0  6.569999999998027
0  6.846099999997383
0  7.12209999999674
0  7.398199999996097
0  7.674299999995453
0  7.95029999999481
0  8.226299999994167
0  8.502399999993523
0  8.77839999999288
0  9.054399999992237
0  9.330399999991593
0  9.60629999999095
0  9.882299999990307
1  1.8802999999998093
1  3.0204000000019495
2  3.0080000000019234
2  3.2530000000024404
2  3.529000000003023
2  3.8052000000036057
2  4.081400000003827
2  4.357600000003183
2  4.6337000000025395
2  4.909900000001896
2  5.186100000001252
2  5.462200000000609
2  5.738299999999965
2  6.014499999999321
2  6.290599999998678
2  6.566699999998034
2  6.842799999997391
2  7.118799999996748
2  7.394899999996104
2  7.670899999995461
2  7.9469999999948175
2  8.222999999994174
2  8.498999999993531
2  8.775099999992888
2  9.051099999992244
2  9.326999999991601
2  9.602999999990958
2  9.878999999990315
```

Data 3: Time points with minimal mutual distances of the bodys, for $\Delta t = 0.0001$. The body is given by the index.

```
0  1.8276400000035058
0  3.115700000011944
0  4.148560000012113
0  6.688749999915947
0  9.12755999982362
1  1.880330000003851
1  3.9731300000175613
1  8.24292999985711
2  3.2026200000125136
2  6.3526099999286725
2  8.890999999832575
```

Data 4: Time points with minimal mutual distances of the bodys, for $\Delta t = 0.00001$. The body is given by the index.