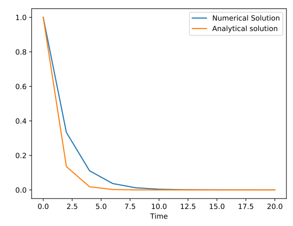
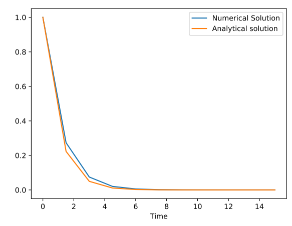
1 - 4th Order Runge Kutta Method

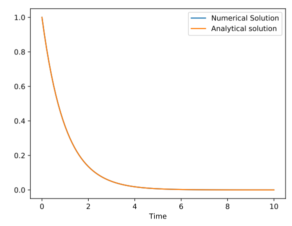
We compare the algorithms solution for different stepsizes h:



(a) $h = 2$



(b) $h = 1.5$



(c) $h = 0.1$

The Numerical approximation seems to converge really quickly towards the analytical solution, even for relatively large stepsizes h.

2 - Three-Body Problem

In order to integrate the Tree-Body Problem a lot of startparameters have to be given to the system.

$$\begin{aligned}
G = m_1 = m_2 = m_3 &= 1 \\
(y_1, y_2) &= -0.97000436, 0.24308753 \\
(y_3, y_4) &= -0.46620368, -0.43236573 \\
(y_5, y_6) &= 0.97000436, -0.24308753 \\
(y_7, y_8) &= -0.46620368, -0.43236573 \\
(y_9, y_{10}) &= 0, 0 \\
(y_{11}, y_{12}) &= 0.93240737, 0.86473146
\end{aligned}$$

The indices of the different starting values are as follows

$$\begin{aligned}
(y_1, y_5, y_9) &: \text{ x-Coordinate of Particles} \\
(y_2, y_6, y_{10}) &: \text{ y-Coordinate of Particles} \\
(y_3, y_7, y_{11}) &: v_x \text{ Initial Velocity of particles in x-Direction} \\
(y_4, y_8, y_{12}) &: v_y \text{ Initial Velocity of particles in y-Direction}
\end{aligned}$$

We store our information of the differential equations in a function `function(y0,x0,m)`. Into this function we feed our $4n$-Dimensional vector y, that contains all the information of our n masses as described above.

The first differential equation we input is simply $f(x_i, y_i) = (v_{x_i}, v_{y_i})$:

```python
def function(y0,x0,m): # This calculates ALL derivatives
    out = np.zeros(y.shape) # y.shape = (12,)
    for index in np.arange(0,len(y0),4): # iterate over the 3 Masses
            out[index] = y0[index+2] # f(x) = vx
            out[index+1] = y0[index+3] # f(y) = vy
    return out
```

What's missing yet is the second differential equation. For that we need to sum over all the forces resulting from other masses in the system. For that we will create a second index, and calculate just those forces, where $i \neq j$. For $\dot{v}_{x_i}$ for example, we get:

$$f(v_{x_i}) = v_{x_i} + G \cdot m_j \cdot \frac{x_j - x_i}{((x_j - x_i)^2 + (y_j - y_i)^2)^{3/2}} = v_{x_i} + G \cdot m_j \cdot \frac{r_x}{|r|^3}$$

Adding this to our `for`-loop, we get:

```python
def function(y0,x0,m): # This calculates ALL derivatives
    out = np.zeros(y.shape) # y.shape = (12,)
    for index in np.arange(0,len(y0),4): # iterate over the 3 Masses
            out[index] = y0[index+2] # f(x) = vx
            out[index+1] = y0[index+3] # f(y) = vy
            for indexx in np.arange(0,len(y0),4):
                if (index != indexx): # sum dynamically over non-equal indizes
                    out[index+2] += G*m[(int)(indexx/4)]*(y0[indexx]-y0[index]
                    )/((y0[indexx]-y0[index])**2+(y0[indexx+1]-y0[index+1]
                    )**2)**1.5 # f(vx) = vx+G*m*(r/|r|*3)

                    out[index+3] += G*m[(int)(indexx/4)]*(y0[indexx+1]-y0[index
                        +1]
                    )/((y0[indexx]-y0[index])**2+(y0[indexx+1]-y0[index+1]
                    )**2)**1.5 # f(vy) = vy+G*m*(r/|r|*3)
    return out
```

Next we write a simple function that stores all of the information provided by the rk4 method nicely, which makes for an easy plot:

```python
def final(h,n,init_vec,m):
    y0,x0 = init_vec
    y0,x0 = rk4(y0,x0,function,h,n,{"m" : m})

    out_x0 = np.zeros(x0.shape)
    out_y0 = np.zeros(x0.shape)
    out_x1 = np.zeros(x0.shape)
    out_y1 = np.zeros(x0.shape)
    out_x2 = np.zeros(x0.shape)
    out_y2 = np.zeros(x0.shape)

    for index, i in enumerate(y0):
        out_x0[index] = i[0]
        out_y0[index] = i[1]
        out_x1[index] = i[4]
        out_y1[index] = i[5]
        out_x2[index] = i[8]
        out_y2[index] = i[9]

    return ((out_x0, out_y0), (out_x1, out_y1), (out_x2, out_y2))
```

Playing around with different stepsizes and runtimes, we can easily get a plot where all the individual particle trajectories are distinguishable. For h = 0.1 and n = 200 we found:

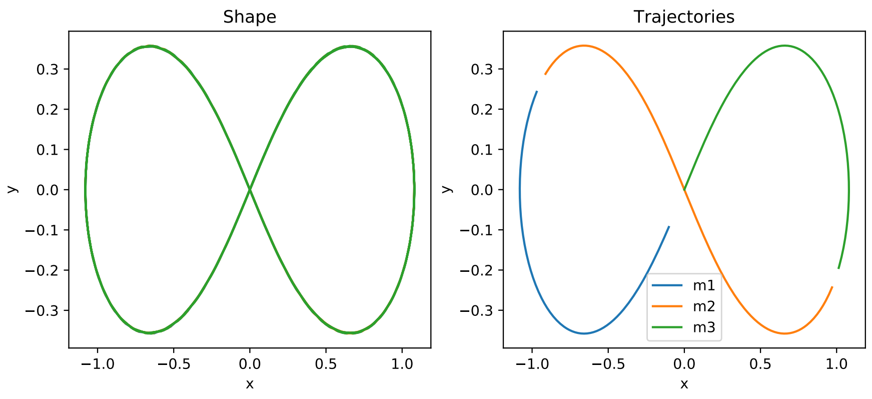


Figure 2: Stable Solution for the 3-Body-Problem

b)

Next, we will look at the Meissel-Burrau Problem for three Masses, with initial velocity of 0:

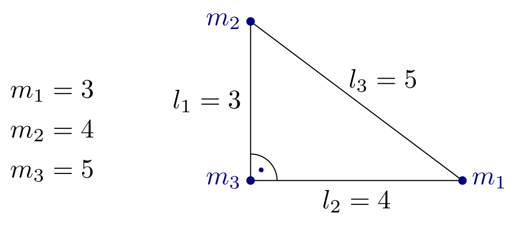


Figure 3: Meissel-Burrau Problem

We transform those coordinates into the CMS system, and run the program for different stepsizes and computation times. We immediately see that choosing smaller stepsizes yields much more accurate interactions between the Masses:

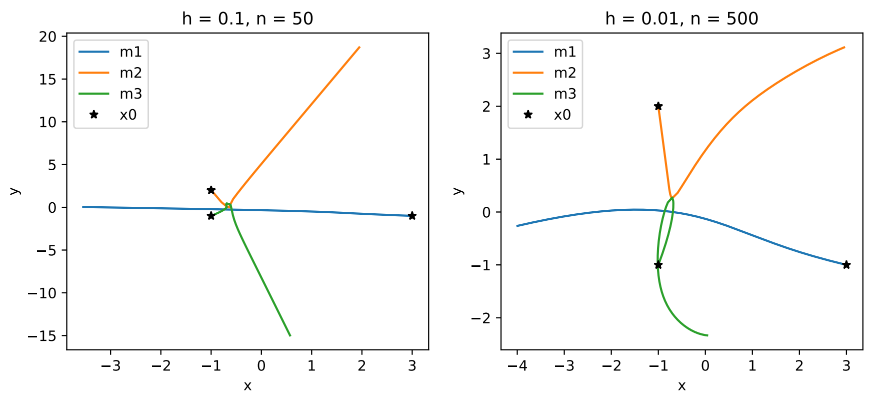


Figure 4: Meissel-Burrau Problem for different Stepsizes

Next, we add a little block extra code to our rk4-loop, that measures the distances of each of the 3 masses. First we need a distance function:

```
    def dist(r1,r2):
        return ((r2[0]-r1[0])**2+(r2[1]-r1[1])**2)**0.5
```

Next we use that to find the minimum distance between the masses, by storing the current distance in a variable, and check with each step, if that variable becomes smaller:

```
def rk4(y0, x0, f, h, n, f_args = {}):
    yn = np.zeros((n+1, y0.shape[0]))
    xn = np.zeros(n+1)
    yn[0,:] = y0
    xn[0] = x0
    mindist12 = 5 # initial distance
    mindist23 = 3 # initial distance
    mindist31 = 4 # initial distance
    mint12 = 0
    mint23 = 0
    mint31 = 0

    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)
        # Calculate distances of masses:
        r1 = np.array([yn[n,0],yn[n,1]])
        r2 = np.array([yn[n,4],yn[n,5]])
        r3 = np.array([yn[n,8],yn[n,9]])

        dist12 = dist(r1,r2)
        dist23 = dist(r2,r3)
        dist31 = dist(r3,r1)

        if (dist12 <= mindist12): # if minimum distance smaller than before,
            save
            mindist12 = dist12
            mint12 = n
        if (dist23 <= mindist23):
            mindist23 = dist23
            mint23 = n
        if (dist31 <= mindist31):
            mindist31 = dist31
            mint31 = n

    print('Minimum␣distances␣for␣h␣=␣', round(h,10), 'and␣n␣=', n)

    print('Minimum␣Distance␣of␣Masses␣1␣and␣2:␣',
            round(mindist12,2), 'at␣t␣=', mint12)
    print('Minimum␣Distance␣of␣Masses␣2␣and␣3:␣',
            round(mindist23,2), 'at␣t␣=', mint23)
    print('Minimum␣Distance␣of␣Masses␣3␣and␣1:␣',
            round(mindist31,2), 'at␣t␣=', mint31, '\n')

    return(yn, xn)
```

In the following table, we examine the minimum distances together with the time at which they occur for different stepsizes h:

h	Distance $\|r\|$	Time t
0.1	$1 \leftrightarrow 2 = 2.65$	21
	$2 \leftrightarrow 3 = 0.15$	21
	$3 \leftrightarrow 1 = 2.15$	24
0.01	$1 \leftrightarrow 2 = 2.20$	270
	$2 \leftrightarrow 3 = 0.03$	188
	$3 \leftrightarrow 1 = 1.76$	310
0.001	$1 \leftrightarrow 2 = 2.99$	1902
	$2 \leftrightarrow 3 = 0.01$	1902
	$3 \leftrightarrow 1 = 2.25$	1964
0.0001	$1 \leftrightarrow 2 = 1.77$	29444
	$2 \leftrightarrow 3 = 0.01$	18793
	$3 \leftrightarrow 1 = 0.54$	30191

Table 1: Minimum mass distances at different stepsizes

From this, we see the problem resulting out of this arrangement of masses. At a certain t, which occurs usually at around $19 \cdot 0.1$ steps, two of the masses get really close, so that v, in some cases, becomes very large. That will result in m_2 and m_3 rapidly slingshotting each other out of the system. This effect is most visible at $h = 0.001$, whereas at $h = 0.0001$ we get a better result again:

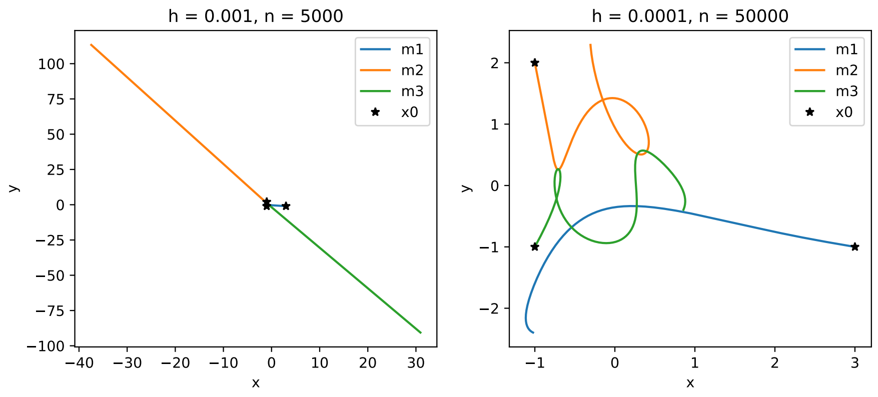


Figure 5: Particle trajectories for small stepsizes h

Finally, we want to plot the particle distance and energy error as functions of time. It is interesting to see, that the Energy error becomes more apparent while approaching $h = 0.001$ but then decreases for even smaller h:

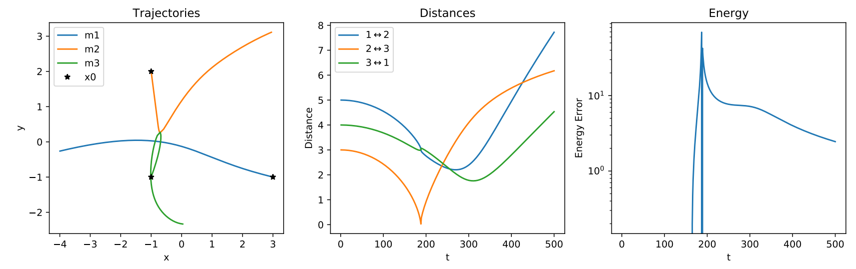


Figure 6: Energy Error for $h = 0.01$ and $n = 500$

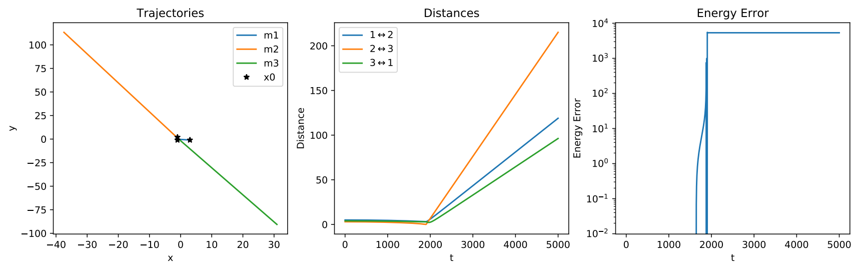


Figure 7: Energy Error for $h = 0.001$ and $n = 5000$

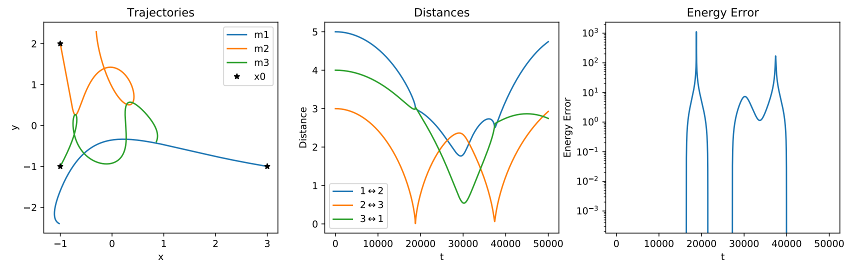


Figure 8: Energy Error for $h = 0.0001$ and $n = 50000$