Course subject: Computational Techniques (2023/2024)
Study course: Electronics and Telecommunications (first level)
University: AGH University of Science and Technology, Kraków, Poland

Topic: Numerical Methods for Solving Ordinary Differential Equations in Two-Body Problem	
Author: Kamil Chaj	Date of submission: 10/01/2024 Supervised by: Przemysław Korohoda, Dr. Eng

Abstract

Two-body problem discuss predicting orbit of two bodies viewed as point masses. The problem assumes that the two objects interact only with one another, and all other bodies are ignored. Goal of this project is to implement and test different methods of numerically solving ordinary differential equations in case of two-body problem. Methods we are going to implement and compare are Euler method, Runge-Kutta method and Adaptive Runge-Kutta method. Methods utilizing multiple evaluations in single iteration like Runge-Kutta method give more accurate results in smaller amount of time than simpler methods like Euler method.

Contents

Chapter 1

Two Body Problem

1.1 Assumptions

For the problem to be not too complicated we can take few assumptions that will simplify solution but maintain overall correct results.

- in system there exists only two bodies
- bodies have uniform mass distributions and are perfectly symmetrical

1.2 Two-body system

In the system we define two bodies with masses m_1, m_2 and positions $\vec{r_1}$, $\vec{r_2}$, where distance between bodies is represented by $\vec{r_0}$ which can be described by following equation.

$$\vec{r_0} = \vec{r_2} - \vec{r_1} \tag{1.1}$$

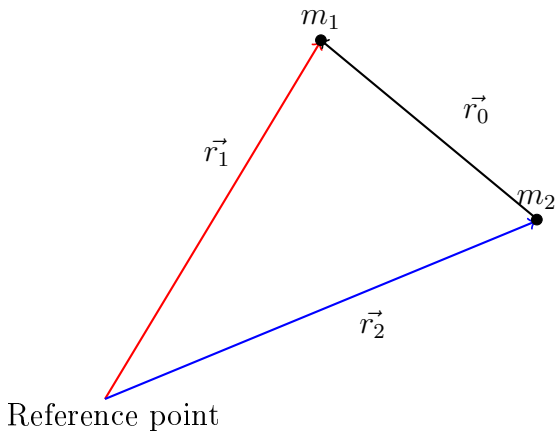


Figure 1.1: two-body system

1.3 Derivation of differential equation

Lets start derivation with defining forces $\vec{F_{12}}$ and $\vec{F_{21}}$ and relation between these forces coming from our assumption that there are only two bodies and therefore only two forces which net force is zero.

$$m_1 \frac{d^2\vec{r_1}}{dt^2} = \vec{F_{12}} \tag{1.2}$$

$$m_2 \frac{d^2\vec{r_2}}{dt^2} = \vec{F_{21}} \tag{1.3}$$

$$\vec{F_{12}} = -\vec{F_{21}} = \vec{F} \tag{1.4}$$

After combining above equations we get system of differential equation describing behavior of any two-body system.

$$\begin{cases} m_1 \frac{d^2\vec{r_1}}{dt^2} = \vec{F} \\ m_2 \frac{d^2\vec{r_2}}{dt^2} = -\vec{F} \end{cases} \tag{1.5}$$

1.4 System of two planets

In our case we want to investigate two-body system where force $\vec{F}$ is described by Newton's law of universal gravitation (1.6)

$$\vec{F} = G\frac{m_1 m_2}{r_0^2}\hat{r_0}, \qquad \hat{r_0} = \frac{\vec{r_0}}{r_0} \tag{1.6}$$

By substituting $\vec{F}$ (1.6) and $\vec{r_0}$ (1.1) in equation (1.5) we get final system of nonlinear differential equations.

$$\begin{cases} \frac{d^2\vec{r_1}}{dt^2} = \frac{Gm_2}{||\vec{r_2} - \vec{r_1}||^3}(\vec{r_2} - \vec{r_1}) \\ \frac{d^2\vec{r_2}}{dt^2} = \frac{Gm_1}{||\vec{r_2} - \vec{r_1}||^3}(\vec{r_1} - \vec{r_2}) \end{cases} \tag{1.7}$$

1.5 Initial conditions

For second order differential equation we need to specify position and velocity at some time, usually at time = 0, to find particular solution. In our system we have two second order equation therefore we need two initial position and velocity vectors which gives us 12 scalar quantities if we consider three dimensional case.

$$\begin{bmatrix} \vec{r_1}(t_0) \\ \vec{r_2}(t_0) \\ \vec{v_1}(t_0) \\ \vec{v_2}(t_0) \end{bmatrix} = \begin{bmatrix} x_1(t_0) & y_1(t_0) & z_1(t_0) \\ x_2(t_0) & y_2(t_0) & z_2(t_0) \\ v_{x1}(t_0) & v_{y1}(t_0) & v_{z1}(t_0) \\ v_{x1}(t_0) & v_{y1}(t_0) & v_{z1}(t_0) \end{bmatrix}, \qquad t_0 = 0 \tag{1.8}$$

implementation in appendix D

1.5.1 Earth-Moon system

First we need to assign values for all parameters in our equations. G parameter is Newtonian constant of gravitation, note that this constant dictates what unites we have to use later

$$G = 6.6743 \cdot 10^{-11} m^3 kg^{-1} s^{-2} \tag{1.9}$$

Mass of the Earth

$$m_1 = 5.97 \cdot 10^{24} kg$$

Mass of the Moon

$$m_2 = 7.34 \cdot 10^{22} kg$$

And initial conditions for the system, $\vec{v_{moon}}$ is perpendicular to position vector, v_{earth} is velocity of Earth and Moon around Sun

$$v_{moon} = 1.022 \cdot 10^3 \frac{m}{s}, \qquad v_{earth} = 29.78 \cdot 10^3 \frac{m}{s}, \qquad r_0 = 3.844 \cdot 10^5 km$$

$$\begin{bmatrix} \vec{v_1}(0) \\ \vec{v_2}(0) \\ \vec{r_1}(0) \\ \vec{r_2}(0) \end{bmatrix} = \begin{bmatrix} 0 & v_{moon} & v_{earth} \\ 0 & 0 & v_{earth} \\ 0 & 0 & 0 \\ r_0 & 0 & 0 \end{bmatrix} \tag{1.10}$$

1.5.2 Projectile motion system

For this case we will look at two-body problem in micro scale, we will verify if assumption that for small distances from surface of the earth gravitational field is uniform.
Gravitation constant G stays the same as in previous case (1.9).

Earth

$$m_1 = 5.97 \cdot 10^{24} kg$$

small body

$$m_2 = 1kg$$

$$v_1 = 0\frac{m}{s}, \qquad v_2 = 5\frac{m}{s}, \qquad R = 6371km, \qquad r = 10m$$

$$\begin{bmatrix} \vec{v_1}(0) \\ \vec{v_2}(0) \\ \vec{r_1}(0) \\ \vec{r_2}(0) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & v_2 & 0 \\ 0 & 0 & 0 \\ R+r & 0 & 0 \end{bmatrix} \tag{1.11}$$

Chapter 2

Numerical method for differential equations

In numerical approximations of differential equation we replace infinitesimal change d with finite change Δ and iteratively calculate small pieces of the curve.

$$\frac{d\vec{v}}{dt} = f(\vec{r}, t) \implies \frac{\Delta\vec{v}}{\Delta t} = f(\vec{r}, t_n) \tag{2.1}$$

$$\begin{aligned} t_n &= t_0 + n\Delta t \\ t_n &= t_{n-1} + \Delta t \end{aligned} \tag{2.2}$$

By multiplying equation (2.1) by Δt we get change of a function in time interval Δt

$$\Delta\vec{v} = f(\vec{r}, t_n)\Delta t \tag{2.3}$$

Truncation error is cause by using finite time step and comes from the definition of limit. Truncation error can be used for adjusting time step in adaptive methods

$$\frac{\Delta\vec{v_{n+1}} - \Delta\vec{v_n}}{\Delta t} = \text{Truncation error} \tag{2.4}$$

2.1 Euler method

This is the simplest method for solving differential equations, where next value is sum of current value and change evaluated at the beginning of Δt

$$\begin{aligned} \vec{r_{n+1}} &= \vec{r_n} + \Delta\vec{r} \\ &= \vec{r_n} + f(\vec{r_n}, t_0 + n\Delta t)\Delta t \end{aligned} \tag{2.5}$$

For higher order equations we simply need to convert n-th order equation to system of n first order equations

$$\begin{bmatrix} \vec{r_{n+1}} \\ \vec{v_{n+1}} \end{bmatrix} = \begin{bmatrix} r_n + v_n\Delta t \\ v_n + f(\vec{r_n}, t)\Delta t \end{bmatrix} \tag{2.6}$$

implementation in appendix D

2.2 4th order Runge-Kutta method

Runge-Kutta method is similar to Euler method but instead of evaluating change of a function once per iteration, in case of 4th order method, it evaluates function 4 times per iteration, and takes weighted average of these evaluations.

$$\begin{aligned} k_1 &= f(\vec{r_n}, t_0 + n\Delta t) \\ k_2 &= f\left(\vec{r_n} + k_1\frac{\Delta t}{2}, t_0 + \Delta t(n + \frac{1}{2})\right) \\ k_3 &= f\left(\vec{r_n} + k_2\frac{\Delta t}{2}, t_0 + \Delta t(n + \frac{1}{2})\right) \\ k_4 &= f\left(\vec{r_n} + k_3\Delta t, t_0 + \Delta t(n + 1)\right) \end{aligned} \tag{2.7}$$

Times at which each evaluation is calculated can be adjusted but the most common is to evaluated function once at beginning and end the interval and twice in the middle of the interval but for second evaluation taking result of the first as an input

$$\vec{v_{n+1}} = \vec{v_n} + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{2.8}$$

Evaluation made in the middle of the interval have usually greater weight.

$$\begin{bmatrix} \vec{r_{n+1}} \\ \vec{v_{n+1}} \end{bmatrix} = \begin{bmatrix} r_n + v_n\Delta t \\ v_n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{bmatrix} \tag{2.9}$$

implementation in appendix D

2.3 Adaptive Runge-Kutta method

In adaptive Runge-Kutta method after each iteration program check if truncation does not exceed set limit, and if it does, evaluation is rejected and the same evaluation is made again with new time step calculated based on maximal and current error until truncation is within set error limit.

For 4th order method new time step is calculated using following formula

$$\Delta\tau = \Delta t\left(\frac{\varepsilon}{e}\right)^{\frac{1}{5}} \tag{2.10}$$

$\Delta\tau$ - new time step, e - truncation error, ε - tolerance

implementation in appendix D

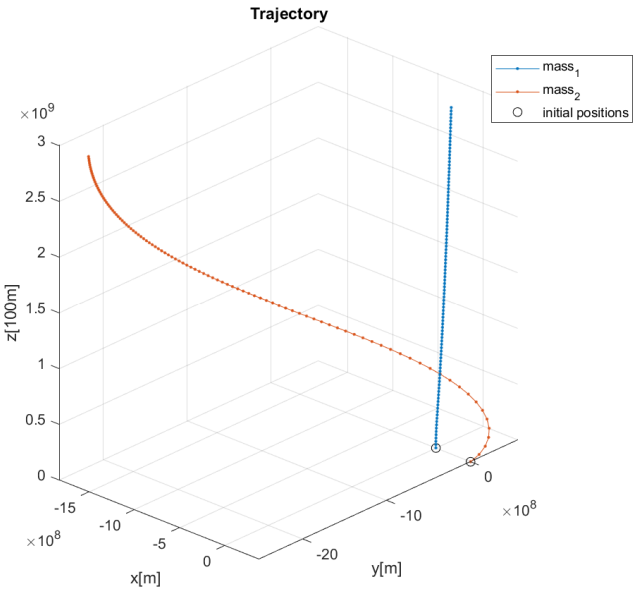
Chapter 3

Results

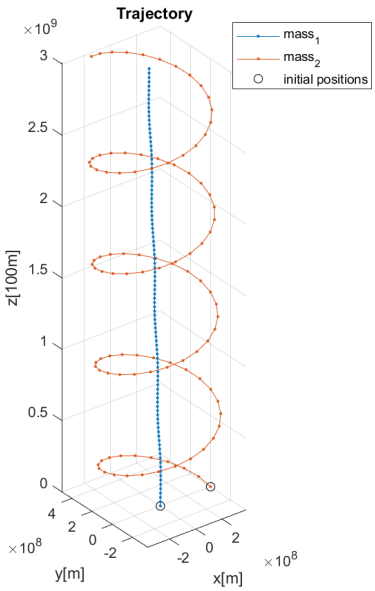
3.1 Trajectory plots

These are Trajectories of Earth-Moon system calculated using methods described in previous chapter. For all calculation we used the same initial conditions, and the same step size for methods with constant time step.

note scale in Z axis is 100m



(a) Trajectory calculated using Euler method



(b) Trajectory calculated using 4th Order Runge-Kutta method

Figure 3.1: Fixed step size

First thing we can notice is that plots differ, but they should not. Difference of solutions comes from differences in the way these method approximates, Euler method evaluates change only once per iteration where Runge-Kutta method evaluated 4 times(in case of 4th order method), at first idea comes to mind to decrease time step for Euler method by 4 times but, it is clear simply decreasing time step is not the best idea (fig. 3.2). Advantage of Runge-Kutta method over Euler method comes from using evaluations calculated during one iteration and plugging the in again to the formula to achieve even more accurate evaluation (eq: 2.7)

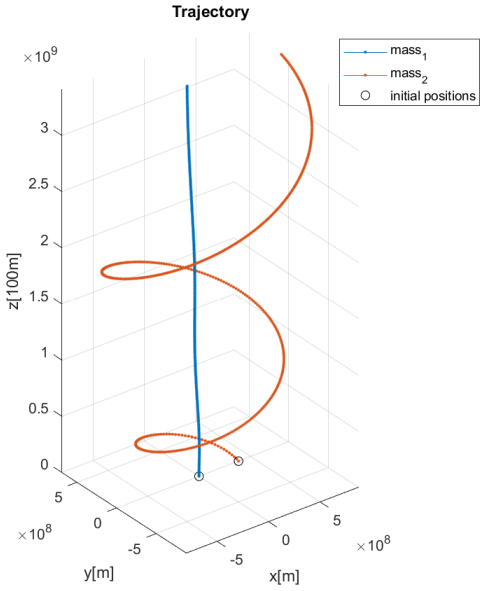


Figure 3.2: Trajectory calculated using Euler method with decrease time step

3.2 Convergence analysis

To check convergence and error of solution depending on used step size or tolerance we need to first expand discrete solution into continuous one with use of interpolation. Interpolated solutions can be then compared. To compare two function we can use cosine similarity, and angular distance.

$$S_C(R_{n-1}, R_n) = \cos(\theta) = \frac{\langle \vec{R_{n-1}}, \vec{R_n} \rangle}{R_{n-1} R_n} \tag{3.1}$$

$$D_\theta(R_{n-1}, R_n) = \arccos(S_C(R_{n-1}, R_n)) = \theta \tag{3.2}$$

Comparing plots for Euler and Runge-Kutta method we can clearly see advantage of Runge-Kutta, which converges for bigger time step and a lot quicker in comparison to Euler method, range in which solution converges partially to real solution is very narrow.

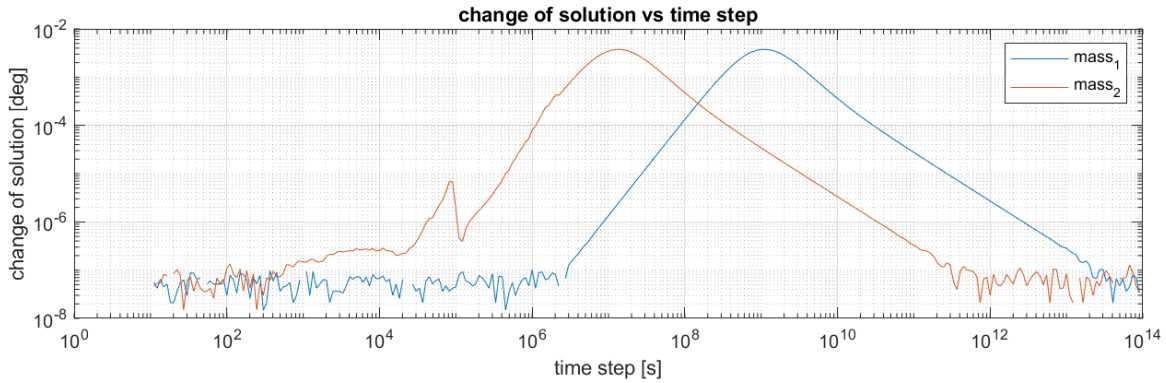


Figure 3.3: Change of solution depending on time step Euler method

Bigger plots in appendix A

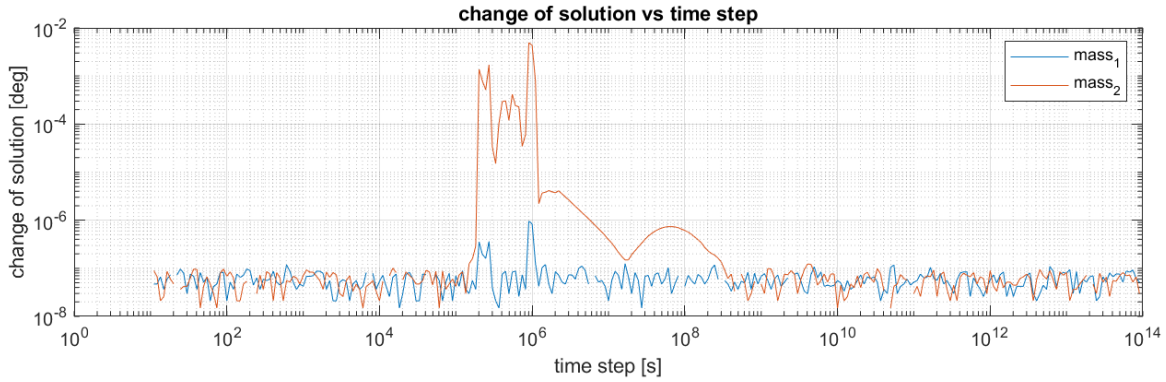


Figure 3.4: Change of solution depending on time step 4th order Runge-Kutta method

3.3 Computation time

To measure computation time and therefore computational complexity we will simply run the calculation for wide range of time steps or tolerances and measure time it took to calculate each trajectory.

It is easy to predict that Euler method will finish computation quicker than Runge-Kutta but when we take into consideration accuracy of the solution, then we can see that we get accurate solution with Runge-Kutta method.

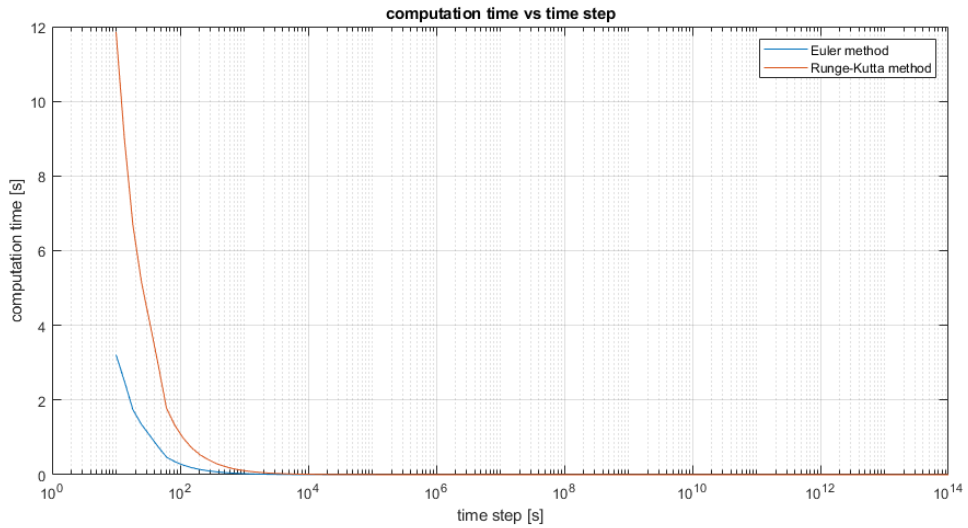


Figure 3.5: Comparison of computation time between methods

From plots in previous section we can assume that for Euler method solution is real when step size is at most magnitude of 10^2 and for Runge-Kutta method 10^5, and when we compare computation times for those time step we can clearly see advantage of Runge-Kutta method.

3.4 Conclusions

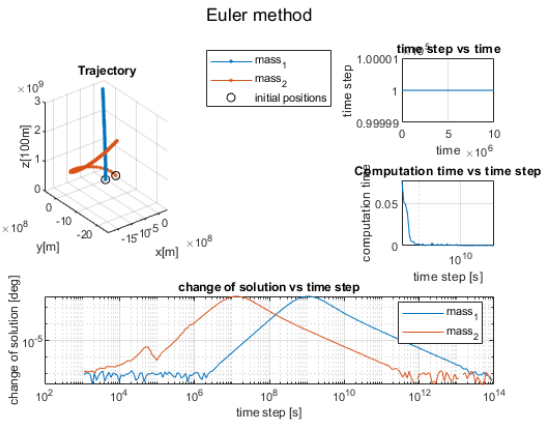
Concluding these test it is clear that using more sophisticated methods for solving differential equations is the better solution than simply using more computational power, by adding few extra steps in numerically solving differential equation we improved greatly accuracy and performance of the method.

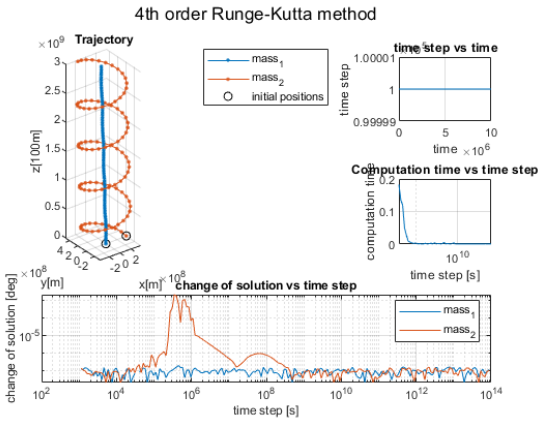
In context of entire project I realized necessity to organize, plan and structure my work before starting, because in small project like this I managed to finish with something that I am satisfied, but I can imagine myself being stuck on one problem or feature and run out of time for the rest of the project.

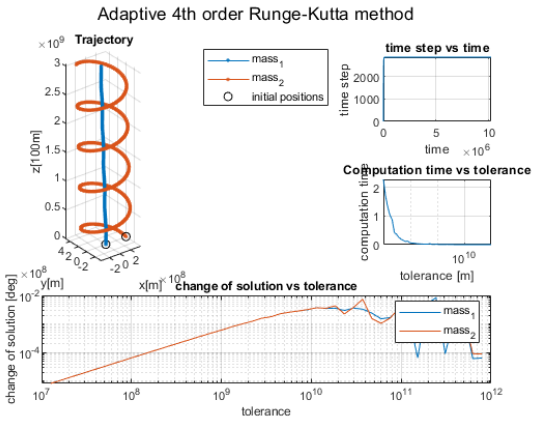
During the project I also extended my abilities to code more efficiently and clearly in MATLAB and better understand MATLAB syntax of matrices, but there is still a lot of room for improvement, because I found myself very often coping and rewriting the same code which is not very good practice.

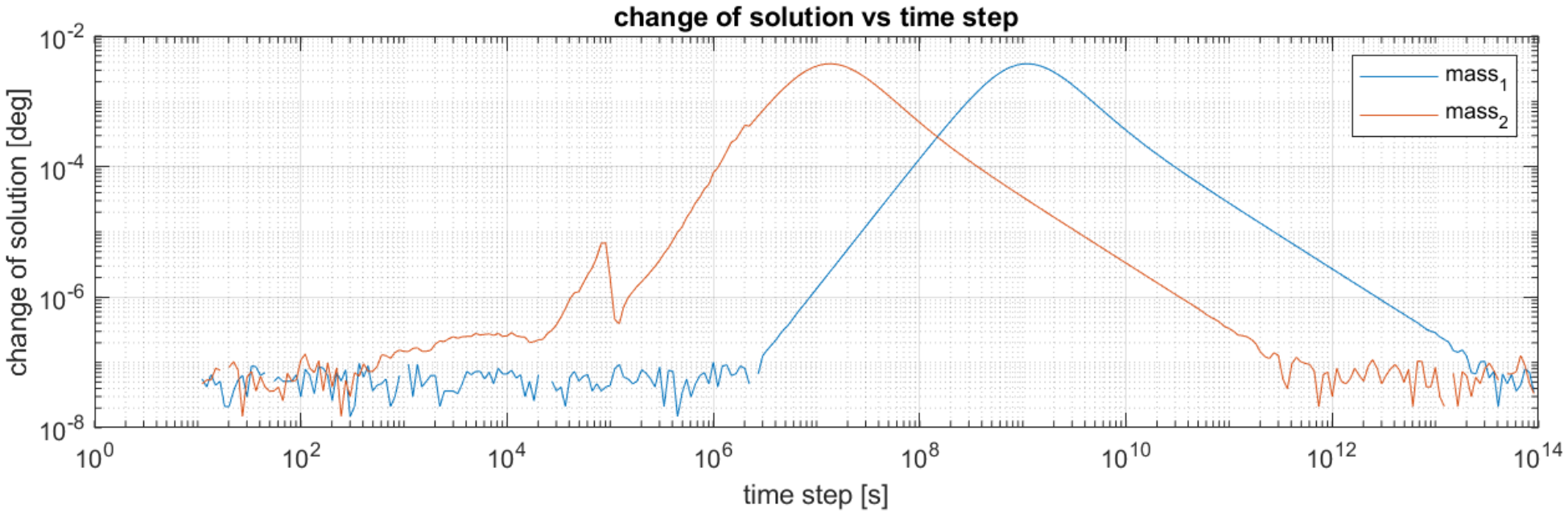
Appendix A

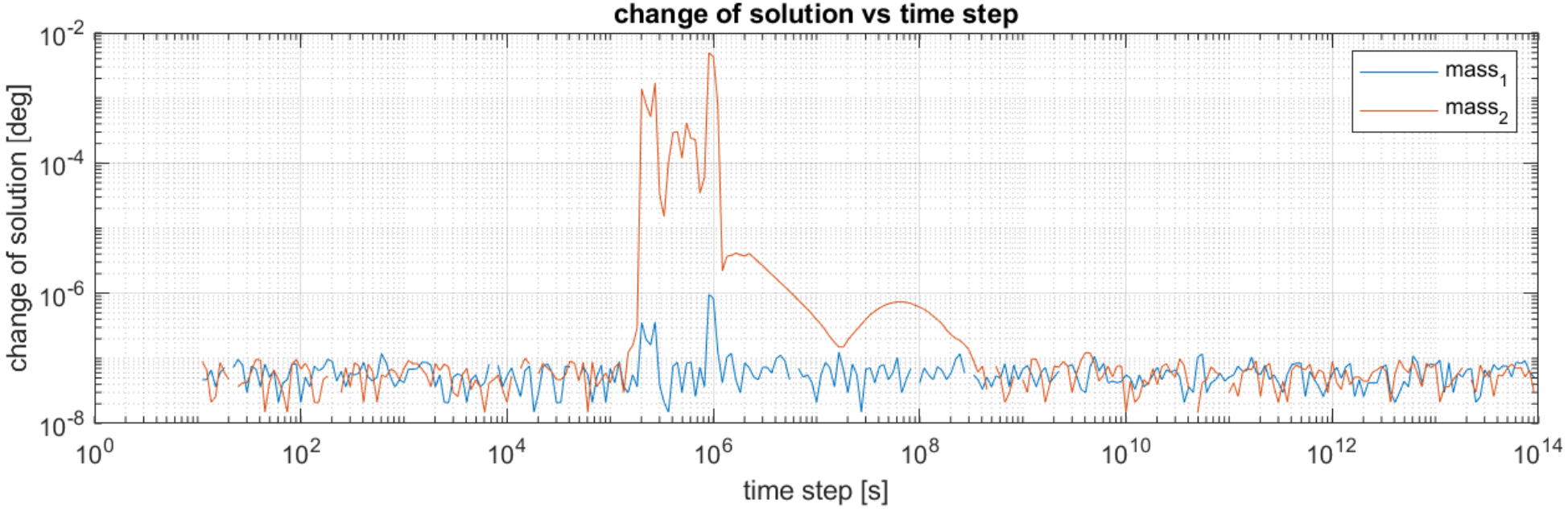
Extra plots








change of solution vs time step
change of solution [deg]
time step [s]
mass1
mass2
10-2
10-4
10-6
10-8
100
102
104
106
108
1010
1012
1014


change of solution vs time step
change of solution [deg]
time step [s]
mass1
mass2
10-2
10-4
10-6
10-8
100
102
104
106
108
1010
1012
1014

Appendix B

Sources

1. GPT 3.5 - Open AI
2. Planetary Fact Sheets - NASA
3. Numerical Methods for Engineers - Jeffrey Chasnov
4. Fundamentals of Orbital Mechanics - Alfonso Gonzalez
5. Wikipedia
   (a) Two-body problem
   (b) Euclidean vector
   (c) Newton's law of universal gravitation
   (d) Runge-Kutta methods
   (e) Truncation error
   (f) Cosine similarity
6. Fundamentals of Physics Extended 10th edition - Halliday & Resnick

Appendix C

Used Matlab functions

C.1 basic functions

- axis
- clc
- clear
- close
- disp
- end
- eps
- figure
- grid
- hold
- length
- linspace
- loglog
- logspace
- plot
- plot3
- semilogx
- size
- subplot
- tic/toc
- title
- xlabel
- ylabel
- zeros

C.2 particular functions

- abs - absolute value
- drawnow - draws figure from buffer
- diff - numerical differentiation
- dot - dot/inner product of a vector
- interp1 - 1D interpolation
- isequal - check if objects are equal
- max - finds maximum value in a vector
- min - finds minimum value in a vector
- norm - calculates norm/magnitude of a vector
- parfor - for loop utilizing parallel computing toolbox
- round - rounds number to integer
- sgtitle - title setting for subplot

Appendix D

Code

Only top files, core functions listed, all code is available on Github repository

Listing D.1: earth_moon.m

```
clc; close all; clear all;

%% Constants
G = 6.67430e-11;

mass = [5.9724e24, 0.07346e24]; %KG
velocity = [29.78e3, 1.022e3]; %M/S
distance = 0.384e9; %m

r1_0 = [0; 0; 0];
r2_0 = [distance; 0; 0];
v1_0 = [0; 0; velocity(1)];
v2_0 = [0; velocity(2); velocity(1)];

initial_conditions = [ v1_0, v2_0, r1_0, r2_0];

period = 708.7 * 60 * 60;
tspan = [0 4*period];
step_size = 1e5;
max_err = 1e8;

%% Euler method
figures2 = Results_plots(@euler, mass, initial_conditions, tspan, step_size↩
    );

%% RK4 method
figures3 = Results_plots(@RK4, mass, initial_conditions, tspan, step_size);

%% Adaptive RK4 method
figures4 = Results_plots(@Adaptive_RK, mass, initial_conditions, tspan, ↩
    max_err);
```

Listing D.2: Results_plots.m

```
function figs = Results_plots(method, mass, initial_conditions, tspan, ↩
    steperr)

     G = 6.67430e-11;
     method_name = ["Adaptive 4th order Runge-Kutta method";
                    "4th order Runge-Kutta method";
                    "Euler method";
                    "none"];
```

```
if isequal(method, @Adaptive_RK) n = 1;
elseif isequal(method, @RK4) n = 2;
elseif isequal(method, @euler) n = 3;
else n = length(method_name) - 1; end

figs = figure("Name", method_name(n));
sgtitle(method_name(n));

tic
subplot(3, 3, [1, 2, 4, 5])
    [t, r] = method(@base_ode, tspan, steperr, initial_conditions, mass, G);
    r(:, [9, 12]) = r(:, [9, 12])./1e2;
    plot3(r(:, [7, 10]), r(:, [8, 11]), r(:, [9, 12]), '.-'); hold on;
    plot3(r(1, [7, 10]), r(1, [8, 11]), r(1, [9, 12]), 'ko'); hold off;
    grid on; axis equal;
    xlabel('x[m]'); ylabel('y[m]'); zlabel('z[100m]');
    title("Trajectory"); legend("mass_1", "mass_2", "initial positions")
toc
disp("Main plot finished")
drawnow


tic
subplot(3, 3, 3)
    dt = diff(t);
    plot(t(1:end-1), dt);
    grid on;
    title("time step vs time")
    xlabel("time"); ylabel("time step")
toc
disp("time step plot finished")
drawnow

tic
subplot(3, 3, 6)
    [test_value, time] = comp_time(method, mass, initial_conditions, tspan);
    semilogx(test_value, time); grid on;
    ylabel("computation time")
    OUT = [test_value; time];
    if ~isequal(method, @Adaptive_RK)
        xlabel("time step [s]")
        title("Computation time vs time step")
    else
        xlabel("tolerance [m]")
        title("Computation time vs tolerance")
    end
toc
disp("computation time plot finished")
drawnow


tic
[test_value, err1, err2] = err_test(method, mass, initial_conditions, tspan);
subplot(3, 3, 7:9)
    loglog(test_value(1:end-1), err1); hold on;
    loglog(test_value(1:end-1), err2); hold on;
```

```
        grid on;
        if ~isequal(method, @Adaptive_RK)
            xlabel("time step [s]"); ylabel("change of solution [deg]")
            title("change of solution vs time step")
        else
            xlabel("tolerance"); ylabel("change of solution [deg]")
            title("change of solution vs tolerance")
        end
        legend("mass_1", "mass_2")
    toc
    disp("error plot finished")

end
```

Listing D.3: base_ode.m

```
function dydt = base_ode(t, r, m_1, m_2, G)
    r_0 = r(10:12) - r(7:9);
    abs_r_0 = norm(r_0);
    vx1 = G .* m_2 .* r_0(1) ./ abs_r_0 .^3;
    vy1 = G .* m_2 .* r_0(2) ./ abs_r_0 .^3;
    vz1 = G .* m_2 .* r_0(3) ./ abs_r_0 .^3;
    vx2 = - G .* m_1 .* r_0(1) ./ abs_r_0 .^3;
    vy2 = - G .* m_1 .* r_0(2) ./ abs_r_0 .^3;
    vz2 = - G .* m_1 .* r_0(3) ./ abs_r_0 .^3;
    dydt(1:6) = [vx1, vy1, vz1, vx2, vy2, vz2];
    dydt(7:12) = r(1:6);
    dydt = dydt';
end
```

Listing D.4: euler.m

```
function [t, r] = euler(func, tspan, h, initial_conditions, mass, G)
    k = 1; t(k) = 0;
    r = zeros(round(tspan(2)/h), 12);
    r(1, :) = initial_conditions(:);
    for k = 2:length(r)
        t(k) = t(k-1) + h;
        r(k, :) = r(k-1, :) + h .* func(0, r(k-1, :), mass(1), mass(2), G)↩
   ';
    end
end
```

Listing D.5: RK4.m

```
function [t, r] = RK4(func, tspan, h, initial_conditions, mass, G)
    k = 1; t(k) = 0;
    r = zeros(round(tspan(2)/h), 12);
    r(1, :) = initial_conditions(:);
    for k = 2:length(r)
        t(k) = t(k-1) + h;
        k_1 = func(0, r(k-1, :), mass(1), mass(2), G)';
        k_2 = func(0, r(k-1, :) + h .* k_1 ./2 , mass(1), mass(2), G)';
        k_3 = func(0, r(k-1, :) + h .* k_2 ./2 , mass(1), mass(2), G)';
        k_4 = func(0, r(k-1, :) + h .* k_3, mass(1), mass(2), G)';
        r(k, :) = r(k-1, :) + h ./ 6 .* (k_1 + 2.*k_2 + 2.*k_3 + k_4);
    end
end
```

Listing D.6: Adaptive_RK.m

```
function [t, r] = Adaptive_RK(func, tspan, max_err, initial_conditions, ↩
    mass, G)
     k = 1; t(k) = 0; h = eps;
     r(1, :) = initial_conditions(:);
     while t < tspan(2)
         k = k + 1;
         while 1
             k_1 = func(0, r(k-1, :), mass(1), mass(2), G)';
             k_2 = func(0, r(k-1, :) + h .* k_1 ./2 , mass(1), mass(2), G)';
             k_3 = func(0, r(k-1, :) + h .* k_2 ./2 , mass(1), mass(2), G)';
             k_4 = func(0, r(k-1, :) + h .* k_3, mass(1), mass(2), G)';
             r(k, :) = r(k-1, :) + h ./ 6 .* (k_1 + 2.*k_2 + 2.*k_3 + k_4);
             e = max([abs(norm(r(k, 7:9)) - norm(r(k - 1, 7:9))), abs(norm(r↩
    (k, 10:12)) - norm(r(k - 1, 10:12)))]);
             h = 0.97 .* h .* (max_err./e).^(1/5);
             if e < max_err
                 break;
             end
         end
         t(k) = t(k-1) + h;
     end
end
```

Listing D.7: Adaptive_RK.m

```
function [test_value, T] = comp_time(method, mass, initial_conditions, ↩
    tspan)
     G = 6.67430e-11;
     % tspan = [0, 1e6];

     if ~isequal(method, @Adaptive_RK)
         test_value = logspace(1, 14, 100);
     else
         test_value = logspace(7, 11, 50);
     end


     parfor k = 1:length(test_value)
         tic
         method(@base_ode, tspan, test_value(k), initial_conditions, mass, G↩
    );
         T(k) = toc;
     end
end
```

Listing D.8: Adaptive_RK.m

```
function [test_value, dr1, dr2] = err_test(method, mass, initial_conditions↩
    , tspan)
     G = 6.67430e-11;
     n = 5e4;

     if ~isequal(method, @Adaptive_RK)
         test_value = logspace(14, 3, 200);
     else
         test_value = logspace(12, 7, 50);
     end

     parfor k = 1:length(test_value)
         [temp1, temp2] = method(@base_ode, tspan, test_value(k), ↩
    initial_conditions, mass, G);
```

```
        [tt, R(:, :, k)] = interpol_arr(temp2, temp1, n);
    end

    parfor k = 2:(length(test_value) - 1)
        k
        norm1 = [norm_arr(R(:, 7:9, k)), norm_arr(R(:, 7:9, k - 1))];
        norm2 = [norm_arr(R(:, 10:12, k)), norm_arr(R(:, 10:12, k - 1))];
        dr1(k) = dot(norm1(:, 1), norm1(:, 2)) ./ (norm(norm1(:, 1)) .* ↩
   norm(norm1(:, 2)));
        dr1(k) = abs(acos(dr1(k)));
        dr2(k) = dot(norm2(:, 1), norm2(:, 2)) ./ (norm(norm2(:, 1)) .* ↩
   norm(norm2(:, 2)));
        dr2(k) = abs(acos(dr2(k)));
    end
end
```