NAWI Graz
Natural Sciences

Bachelor thesis

Numerical simulation of the planetary motions in the solar system with Runge Kutta Methods

by
Alexander Josef Seyr

supervised by
Assoc.- Prof. Dr. Peter Puschnig

University of Graz, Austria

Institute of Physics

Department of Theoretical Physics

November 6, 2020

Abstract

The origins of modern astronomy lie within the planetary system we call home, the Solar System. It is located in an outer spiral arm of the Milky Way galaxy and consists of our star, the Sun, and everything else bound to it by gravity [15],[18]. To unravel the dynamics and evolution of the Solar System we must understand the law of gravitation, which in this sense was the key discovery of Newton and explains the orbits of planets, satellites and comets, so their future position can be predicted [2]. Newton's theory of gravity has its limits and the current model of gravity is given by Einstein's general theory of relativity, which since its development withstood every experimental test. However, for the purpose of this thesis, Newton's theory of gravity is sufficient [8]. Observing, exploring and ultimately understanding our solar system was the first step towards understanding more of the universe. Many breakthroughs in the dynamics of the Solar System have resulted from the use of computers [2]. Thus in this thesis, by utilizing the theory a set of differential equations will be derived, which can then be solved numerically with given initial values. The resulting initial value problem will be solved with the famous Runge Kutta methods to analyze their solutions and to show how chosen methods fit this specific problem. In the beginning some background material will be presented followed by the required theory and a discussion of the implementation. Finally the results of the simulation will be discussed.

Here I would also like to point out that this thesis was inspired by a graduate project of the Western Illinois University in mathematics and philosophy, "The equations for planetary motion and their numerical solution", written by Jonathan Njeunje and Dinuka Sewwandi [11]. In their project, using mainly Matlab, they focus on the 4th order Runge Kutta method, verify its order of convergence and stability. Whereas this paper, using Python, will focus on a comparison of 4th order Runge-Kutta method and the Runge-Kutta-Fehlberg method.

Contents

1 Introduction **3**
1.1 Solar System . . . 3
1.2 Newton's law of gravity . . . 3

2 Initial value problem **4**

3 Runge Kutta Methods **7**
3.1 The Runge-Kutta method 4th order . . . 9
3.2 The Runge-Kutta-Fehlberg method . . . 10
3.2.1 Stepsize Control . . . 10

4 Implementation **11**
4.1 Verification of the initial value problem . . . 11
4.2 Implementation of the Runge-Kutta method 4th order . . . 16
4.3 Implementation of the Runge-Kutta-Fehlberg method . . . 16
4.4 Error estimation . . . 17

5 Results and discussion **18**
5.1 Runge-Kutta method 4th order . . . 18
5.2 Runge-Kutta-Fehlberg method . . . 21
5.3 Performance . . . 24

6 Summary **25**

A Python code **29**
A.1 Initial value problem . . . 29
A.2 Implemented Runge-Kutta-Fehlberg method . . . 29

1 Introduction

1.1 Solar System

Our Solar System consists of our star, the Sun, the planets Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus and Neptune, dwarf planets such as Pluto (illustrated in Fig.1), dozens of moons and millions of asteroids, comets and meteoroids [15].

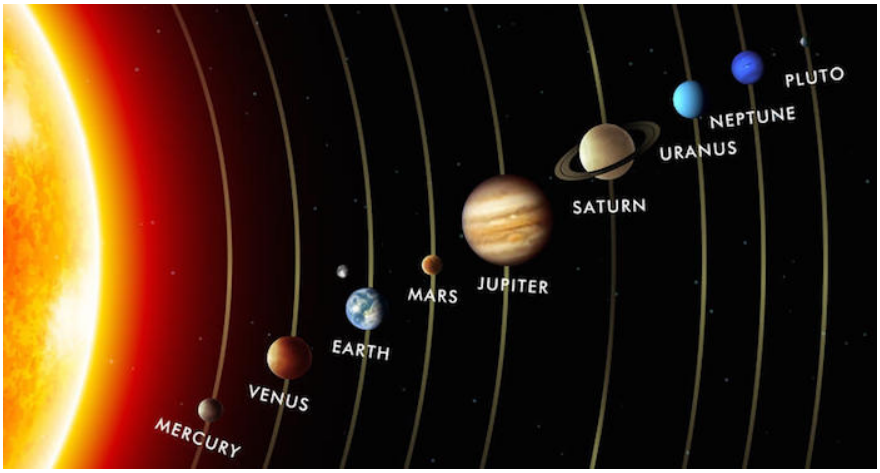


Figure 1: Illustration of the Solar System [6]

All the planets move in stable elliptic trajectories around the sun, and the sun describes its own motion with respect to the center of mass of the system [9]. Pluto is one of the planets classified as a dwarf planet. This class includes all massive round objects, which orbit the Sun, but haven not cleared their orbital path [15]. These motions are governed by Kepler's Laws of planetary motion, which are a result of Newton's law of gravity and conservation of angular momentum and energy. Even though the systems behavior is well understood, it still has chaotic characteristics. In the sense, that in theory the system is perfectly predictable over time, but to solve it, initial conditions are needed. Since these initial conditions have to be measured, they always contain a small error which can accumulate to large errors in the long term behavior [18].

1.2 Newton's law of gravity

The celestial mechanics for the purpose of this thesis are best described by Newton's law of planetary gravitation. To apply this law, the assumption of planet masses being approximated as point masses needs to be made, which is perfectly viable due to the vast distances between all the bodies in the Solar System. Via utilizing Newton's law of gravitation the force $\boldsymbol{F}_{ik}$ of the point mass m_i acting on another point mass m_k located at $\boldsymbol{r}_i$ and $\boldsymbol{r}_k$ is given by

$$\mathbf{F}_{ik} = Gm_i m_k \frac{\mathbf{r}_i - \mathbf{r}_k}{|\mathbf{r}_i - \mathbf{r}_k|^3}, \tag{1}$$

where G is the gravitational constant.

In Cartesian coordinates the components of Eq. 1 are:

$$F_{ik_x} = |\mathbf{F}_{ik}|\frac{x_i - x_k}{r_{ik}} \tag{2}$$

$$F_{ik_y} = |\mathbf{F}_{ik}|\frac{y_i - y_k}{r_{ik}} \tag{3}$$

$$F_{ik_z} = |\mathbf{F}_{ik}|\frac{z_i - z_k}{r_{ik}} \tag{4}$$

$$r_{ik} = \sqrt{(x_i - x_k)^2 + (y_i - y_k)^2 + (z_i - z_k)} \tag{5}$$

By applying Newton's second law of dynamics, $F = m\frac{d^2r}{dt^2}$ on Eqs. 2-4 we get:

$$\frac{d^2x_i}{dt^2} = Gm_k\frac{x_i - x_k}{r_{ik}^3} \tag{6}$$

$$\frac{d^2y_i}{dt^2} = Gm_k\frac{y_i - y_k}{r_{ik}^3} \tag{7}$$

$$\frac{d^2z_i}{dt^2} = Gm_k\frac{z_i - z_k}{r_{ik}^3} \tag{8}$$

This set of coupled second order ordinary differential equations describes the motion of the mass m_i bound to the mass m_k through gravity [8].

The next step will be to generalize this set of equations for our purpose of planetary motions of the Solar System to define the initial value problem, on which we then can apply the chosen Runge-Kutta methods. The goal is to create numerically stable methods and to analyze their behavior, when applied on this problem.

2 Initial value problem

An initial value problem is defined by its differential equation,

$$y^{(n)} = f(t, y(t), ..., y^{(n-1)}(t)), \qquad t \ \epsilon \ I \tag{9}$$

and a set of n initial conditions,

$$y(t_0) = y_0, \quad y'(t_0) = y_1, ..., y^{(n-1)}(t_0) = y_{n-1} \tag{10}$$

The number of initial conditions equals the order of the differential equation and therefore also the number of integration constants of the solution. Since every n-th order differential equation can be reformulated as a set of coupled first order differential equations we get a system with n equations and n unknowns [19].

Due to the fact that every object in the solar system is interacting with each other, we have to generalize Eqs. 2-4 for N objects with masses $m_1, m_2..., m_i, ..., m_N$ acting on one chosen object k with mass m_k, which results in the following system of equations.

$$\frac{d^2x_k}{dt^2} = \sum_{i=1;i\neq k}^{N} Gm_i \frac{x_i - x_k}{r_{i,k}^3} \tag{11}$$

$$\frac{d^2y_k}{dt^2} = \sum_{i=1;i\neq k}^{N} Gm_i \frac{y_i - y_k}{r_{i,k}^3} \tag{12}$$

$$\frac{d^2z_k}{dt^2} = \sum_{i=1;i\neq k}^{N} Gm_i \frac{z_i - z_k}{r_{i,k}^3} \tag{13}$$

$$r_{ik} = \sqrt{(x_i - x_k)^2 + (y_i - y_k)^2 + (z_i - z_k)^2} \tag{14}$$

With r_{ik} being the distance between object i and object k [7]. Now we can reformulate Eqs. 11-13 into a standard initial value problem [19].

$$\mathbf{Y} \equiv \begin{pmatrix} x \\ y \\ z \\ \frac{dx}{dt} \\ \frac{dy}{dt} \\ \frac{dz}{dt} \end{pmatrix} \equiv \begin{pmatrix} x \\ y \\ z \\ v_x \\ v_y \\ v_z \end{pmatrix} \tag{15}$$

$$\frac{d\mathbf{Y}}{dt} = \frac{d}{dt}\begin{pmatrix} x \\ y \\ z \\ v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ \sum_{i=1;i\neq k}^{N} Gm_i \frac{x_i - x_k}{r_{i,k}^3} \\ \sum_{i=1;i\neq k}^{N} Gm_i \frac{y_i - y_k}{r_{i,k}^3} \\ \sum_{i=1;i\neq k}^{N} Gm_i \frac{z_i - z_k}{r_{i,k}^3} \end{pmatrix} \tag{16}$$

Finally for being able to solve our system of $2N$ coupled first order differential equations 16, we need a set of initial conditions for all the objects of the solar system we want to include in our calculations.
For this purpose the initial conditions were obtained by using "JPL Horizons On-Line Ephemeris System" accessed via the offered web-interface [1]. Chosen were the eight planets orbiting our sun and the dwarf planet Pluto. All of their initial coordinates, velocities, mass and radii are listed in Tab. 1, which represents the state of these objects on April 6, 2018.

Table 1: Initial conditions of the chosen objects from the April 6, 2018.
x, y, z ... Cartesian coordinates in meters
v_x, v_z, v_z ... respective velocities in meters per second
m ... mass in kilograms
r ... radius in meters

#	OBJECT	x / m	y / m	z / m	$\mathbf{v_x}$ / $\frac{\mathbf{m}}{\mathbf{s}}$	$\mathbf{v_y}$ / $\frac{\mathbf{m}}{\mathbf{s}}$	$\mathbf{v_z}$ / $\frac{\mathbf{m}}{\mathbf{s}}$	m / kg	r / m
1	SUN	1.81899e+08	9.83830e+08	-1.58778e+07	-1.12474e+01	7.54876e+00	2.68723e-01	1.98854e+30	6.95500e+8
2	MERCURY	-5.67576e+10	-2.73592e+10	2.89173e+09	1.16497e+04	-4.14793e+04	-4.45952e+03	3.30200e+23	2.44000e+06
3	VENUS	4.28480e+10	1.00073e+11	-1.11872e+09	-3.22930e+04	1.36960e+04	2.05091e+03	4.86850e+24	6.05180e+06
4	EARTH	-1.43778e+11	-4.00067e+10	-1.38875e+07	7.65151e+03	-2.87514e+04	2.08354e+00	5.97219e+24	6.37101e+06
5	MARS	-1.14746e+11	-1.96294e+11	-1.32908e+09	2.18369e+04	-1.01132e+04	-7.47957e+02	6.41850e+23	3.38990e+06
6	JUPITER	-5.66899e+11	-5.77495e+11	1.50755e+10	9.16793e+03	-8.53244e+03	-1.69767e+02	1.89813e+27	6.99110e+07
7	SATURN	8.20513e+10	-1.50241e+12	2.28565e+10	9.11312e+03	4.96372e+02	-3.71643e+02	5.68319e+26	5.82320e+07
8	URANUS	2.62506e+12	1.40273e+12	-2.87982e+10	-3.25937e+03	5.68878e+03	6.32569e+01	8.68103e+25	2.53620e+07
9	NEPTUNE	4.30300e+12	-1.24223e+12	-7.35857e+10	1.47132e+03	5.25363e+03	-1.42701e+02	1.02410e+26	2.46240e+07
10	PLUTO	1.65554e+12	-4.73503e+12	2.77962e+10	5.24541e+03	6.38510e+02	-1.60709e+03	1.30700e+22	1.19500e+06

3 Runge Kutta Methods

To solve the initial value problem, two methods were chosen, the classic Runge-Kutta method 4th order and the Fehlberg method or Runge-Kutta 4(5) method. Before discussing these two methods some general information about the family of Runge-Kutta methods will be covered.
For the purpose of illustrating the basic concept, $n = 1$ is used in Eq. 9 and 10. For reasons explained in Sec. 2, we just have to look at one ordinary differential equation:

$$\dot{y}(t) = f(y, t) \tag{17}$$

with an initial value:

$$y(0) = y_0 \tag{18}$$

By integrating Eq. 17 over the interval $[t_n, t_{n+1}]$, we get:

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} dt' f[y(t'), t'] \tag{19}$$

Eq. 19 is still exact and the first step towards a numeric solver for initial value problems is to approximate the included integral. The error made by such a local truncation in a single step can be classified by the highest order of the Taylor series, where the approximation is still exact, therefore the capital $\mathcal{O}$ notation will be used. For further information see, for instance Ref. [4]

Now the idea of the Runge Kutta methods is to improve the accuracy of the approximated integral by calculating intermediate grid points within the interval $[t_n, t_{n+1}]$ and therefore reduce the local error by matching more terms with the Taylor series expansion of the function $f(y, t)$. The simplest method which illustrates this idea is the central rectangular rule, where the intermediate grid point $f(y_{n+\frac{1}{2}}, t_{n+\frac{1}{2}})$, with $t_{n+\frac{1}{2}} = t_n + \frac{\Delta t}{2}$, is taken into account. Then Eq. 19 results in the approximation:

$$y_{n+1} = y_n + f(y_{n+\frac{1}{2}}, t_{n+\frac{1}{2}})\Delta t + \mathcal{O}(\Delta t^3) \tag{20}$$

Due to its $two - step$ characteristic we now have to find an approximation for $y_{n+\frac{1}{2}}$, which will be done here with the explicit Euler method and the average function value. First, $y_{n+\frac{1}{2}}$ is approximated with the explicit Euler method, which results in:

$$y_{n+\frac{1}{2}} = y_n + \frac{\Delta t}{2} f(y_n, t_n) + \mathcal{O}(\Delta t^2) \tag{21}$$

By inserting Eq. 21 into Eq. 20 we obtain:

$$y_{n+1} = y_n + f(y_n + \frac{\Delta t}{2} f(y_n, t_n), t_{n+\frac{1}{2}})\Delta t + \mathcal{O}(\Delta t^2) \tag{22}$$

Eq. 22 is also called the explicit midpoint rule.

Second, $y_{n+\frac{1}{2}}$ is approximated with the average function value of the interval. This, when inserted in Eq.20 gives us:

$$y_{n+1} = y_n + f(\frac{y_n + y_{n+1}}{2}, t_n + \frac{\Delta t}{2})\Delta t + \mathcal{O}(\Delta t^2) \tag{23}$$

Also referred to as implicit midpoint rule.

For building the bridge to a Runge-Kutta method of stage d, with our examples, an algorithmic form of notation will be introduced. This will clarify the sequence of terms which should be calculated in a specific order. It can be achieved by defining some variables Y_i with $i \geq 1$ to which intermediate solutions are assigned. In the following this notation will be illustrated on our examples discussed above.

First on the explicit Euler method Eq.21, which we obviously can also use to approximate the integral in Eq.19, which then has the form,

$$y_{n+1} = y_n + \Delta t f(y_n, t_n) \tag{24}$$

and with the described notation we get:

$$\begin{aligned} Y_1 &= y_n \\ y_{n+1} &= y_n + \Delta t f(Y_1, t_n) \end{aligned} \tag{25}$$

second on the explicit midpoint rule Eq.22:

$$\begin{aligned} Y_1 &= y_n, \\ Y_2 &= y_n + \frac{\Delta t}{2} f(Y_1, t_n + \frac{\Delta t}{2}), \\ y_{n+1} &= y_n + \frac{\Delta t}{2} f(Y_2, t_n + \frac{\Delta t}{2}) \end{aligned} \tag{26}$$

and third on the implicit midpoint rule Eq.23, which can then be written as:

$$\begin{aligned} Y_1 &= y_n + \frac{\Delta t}{2} f(Y_1, t_n + \frac{\Delta t}{2}), \\ y_{n+1} &= y_n + \Delta t f(Y_1, t_n + \frac{\Delta t}{2}) \end{aligned} \tag{27}$$

These representations Eq. 25, Eq.26 and Eq.27 are all specific cases of the general form of a d-stage Runge Kutta method:

$$\begin{aligned} Y_i &= y_n + \Delta t \sum_{j=1}^{d} a_{ij} f(Y_j, t_n + c_j \Delta t), \qquad i = 1, ..., d, \\ y_{n+1} &= y_n + \Delta t \sum_{j=1}^{d} b_j f(Y_j, t_n + c_j \Delta t) \end{aligned} \tag{28}$$

Eq. 28 is fully determined by the coefficients a_{ij}, which forms a $d \times d$ matrix, whereas b_j and c_j are d dimensional vectors. The conventional way to specify such methods is through Butcher tableau's.

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1d} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_d & a_{d1} & a_{d2} & ... & a_{dd} \\ \hline & b_1 & b_2 & ... & b_d \end{array} \tag{29}$$

The Butcher tableau's for the above methods are:

$$\textit{explicit Euler} \qquad \begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \tag{30}$$

$$\textit{explicit midpoint} \qquad \begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \tag{31}$$

$$\textit{implicit midpoint} \qquad \begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array} \tag{32}$$

This also makes it easier to identify whether a Runge-Kutta method is explicit or implicit, because if the matrix a_{ij} is zero on and above the diagonal, it is an explicit method, while otherwise it is an implicit method. [3],[4],[20],[12]

3.1 The Runge-Kutta method 4th order

Probably the most popular scheme among the Runge-Kutta methods is the classical 4-stage Runge-Kutta method. It is defined by the algorithm:

$$\begin{aligned} Y_1 &= y_n \\ Y_2 &= y_n + \frac{\Delta t}{2} f(Y_1, t_n) \\ Y_3 &= y_n + \frac{\Delta t}{2} f(Y_2, t_n + \frac{\Delta t}{2}) \\ Y_4 &= y_n + \Delta t f(Y_3, t_n + \frac{\Delta t}{2}) \\ y_{n+1} &= y_n + \frac{\Delta t}{6}[f(Y_1, t_n) + 2f(Y_2, t_n + \frac{\Delta t}{2}) + 2f(Y_3, t_n + \frac{\Delta t}{2}) + f(Y_4, t_n)] \end{aligned} \tag{33}$$

or just by the Butcher tableau:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array} \tag{34}$$

Eq. 34 indicates, that it is an explicit Runge Kutta Method and due to its 4-stages it can be shown to have an local truncation error of order $\mathcal{O}(\Delta t^5)$. [3],[20]

3.2 The Runge-Kutta-Fehlberg method

The Fehlbergmethod or Runge Kutta Fehlberg 4(5) Method will be used for implementing a method with adaptive step size or rather error control, which will be described in the next section.

The Butcher tableau describing this method is the following:

$$\begin{array}{c|cccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1/4 & 1/4 & 0 & 0 & 0 & 0 & 0 \\ 3/8 & 3/32 & 9/32 & 0 & 0 & 0 & 0 \\ 12/13 & 1932/2197 & -7200/2197 & 7296/2197 & 0 & 0 & 0 \\ 1 & 439/216 & -8 & 3680/513 & -845/4104 & 0 & 0 \\ 1/2 & -8/27 & 2 & -3544/2565 & 1859/4104 & -11/40 & 0 \\ \hline & 16/135 & 0 & 6656/12825 & 28561/56430 & -9/50 & 2/55 \\ & 25/216 & 0 & 1408/2565 & 2197/4104 & -1/5 & 0 \end{array} \tag{35}$$

The algorithmic notation of this method will not be presented due to its complexity, but can be written out by utilizing Eq. 28. This method gives two solutions for each step. Where the penultimate row of the Butcher tableau, Eq. 35, represents a fifth order solution and the last row represents a fourth order solution [5, 13, 20].

3.2.1 Stepsize Control

The common approach to create an algorithm with adaptive step size and therefore local error control with the Fehlberg method is to introduce a scaling factor q, which is used to keep the local error in a desired error tolerance ϵ. Since we get two approximations with different errors, where $\tilde{y}_{n+1}$ represents an approximation of 5-th order and y_{n+1} represents an approximation of 4-th order, see Eq. 35, we can use them to calculate the error made each step. Both of these approximations are of the form Eq. 28. Let us denote the exact function value as $y(t_{n+1})$, then we can write the local truncation error of these methods as:

$$\tau_{n+1}(\Delta t) = \frac{y(t_{n+1}) - y_{n+1}}{\Delta t} \tag{36}$$

$$\tilde{\tau}_{n+1}(\Delta t) = \frac{y(t_{n+1}) - \tilde{y}_{n+1}}{\Delta t} \tag{37}$$

By subtracting Eq.36 from Eq. 37 we get

$$\tau_{n+1}(\Delta t) = \tilde{\tau}_{n+1}(\Delta t) + \frac{1}{\Delta t}(\tilde{y}_{n+1} - y_{n+1}), \tag{38}$$

where $\tilde{\tau}_{n+1}(\Delta t)$ can be neglected, because the significant portion of the error comes from τ_{n+1}. This offers us an easy way to compute approximation for the local truncation error of the 4-stage method:

$$\tau_{n+1} \approx \frac{1}{\Delta t}(\tilde{y}_{n+1} - y_{n+1}) \tag{39}$$

To find out how to modify Δt that the local truncation error is approximately equal to a prescribed tolerance ϵ, we first assume that since $\tau_{n+1}(\Delta t)$ is $\mathcal{O}(\Delta t^4)$ a number K independent of Δt exists, with

$$\tau_{n+1}(\Delta t) \approx K \Delta t^4 \tag{40}$$

Then by replacing Δt with $q\Delta t$, where q describes a scaling factor, we get:

$$\tau_{n+1}(q\Delta t) \approx K(q\Delta t)^4 \approx q^4\tau_{n+1}(\Delta t) \approx \frac{q^4}{h}(\tilde{y}_{n+1} - y_{n+1}) \tag{41}$$

To bind $\tau_{n+1}(q\Delta t)$ by ϵ, we need to choose q so that

$$\frac{q^4}{h}(\tilde{y}_{n+1} - y_{n+1}) \approx \tau_{n+1}(q\Delta t) \leq \epsilon \tag{42}$$

and therefore

$$q \leq \left(\frac{\epsilon\Delta t}{|\tilde{y}_{n+1} - y_{n+1}|}\right)^{1/4} \tag{43}$$

Now we have the scaling factor which lets us control the time steps to stay near a desired error tolerance ϵ, to implement the Fehlberg method with adaptive stepsize control. [13, 5]

4 Implementation

For a numerical implementation the programming language *Python* 3.7 was used with the scientific virtual environment *Spyder* 4.1.1.

4.1 Verification of the initial value problem

Initially there had to be checked, if the initial value problem, Eq. 16 with the initial values Tab. 1, was designed correctly. Therefore the function describing the initial value problem, for implementation see A.1, was tested with two built in methods, both are included within the *SciPy* (scientific python) software package and part of the sub module *Integration*, which can be accessed via `scipy.integration`.

The first built in method used for the verification, was a solver which is accessed through `scipy.integrate.odeint`. It utilizes LSODA from the FORTRAN library odepack, which can solve stiff and nonstiff systems [16]. By applying this method for a time span of 250 years, which is roughly the orbital period of Pluto, we obtain the results shown in Fig.2 and 3. We see, that all planets form stable orbits for this time span, which indicates that this method was able to solve our initial value problem without any instabilities.

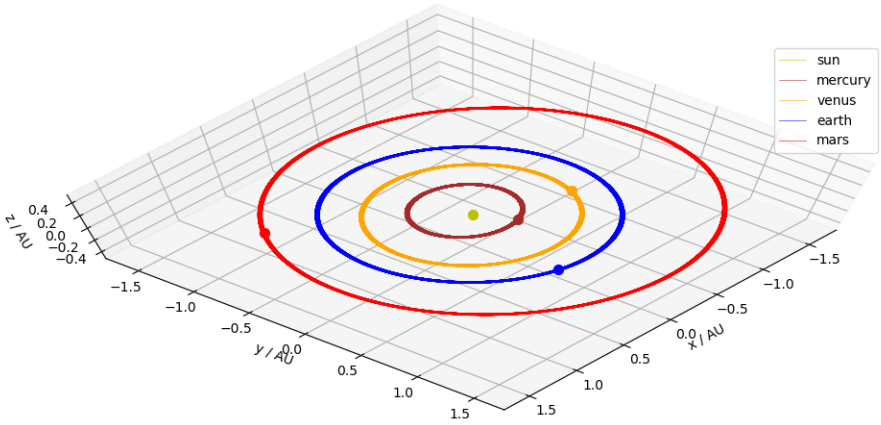


Figure 2: Orbits inner planets, using `scipy.integrate.odeint` for 250 years

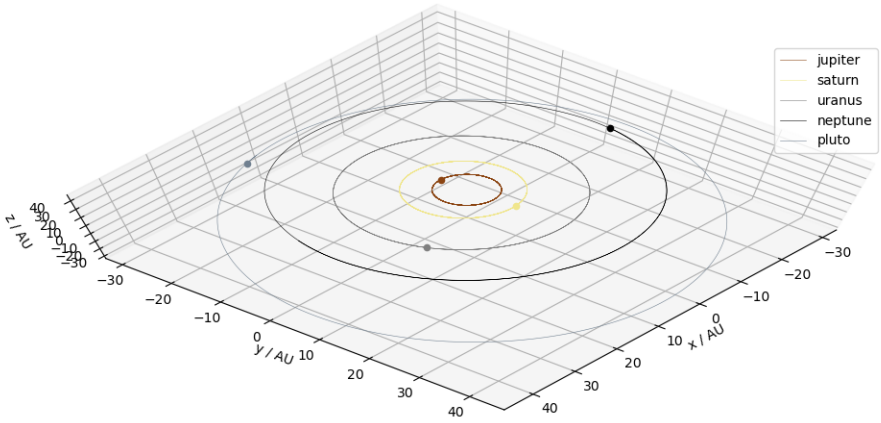
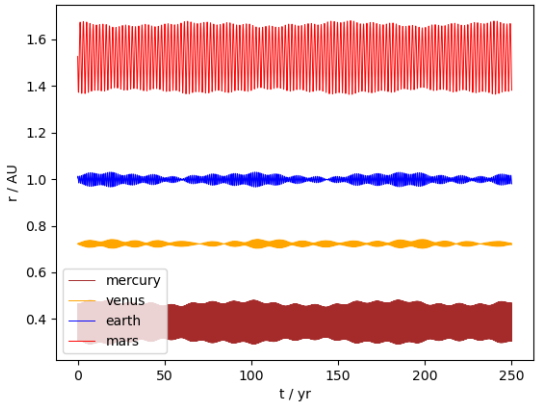
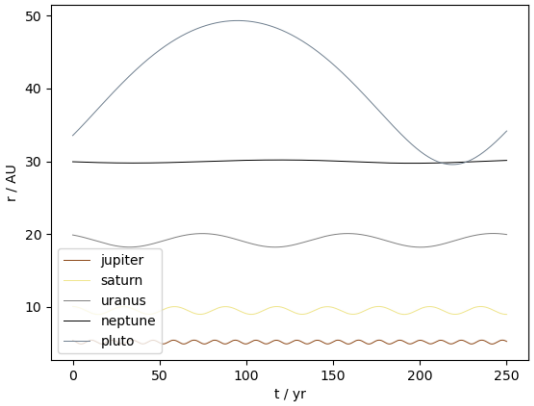


Figure 3: Orbits outer planets, using `scipy.integrate.odeint` for 250 years

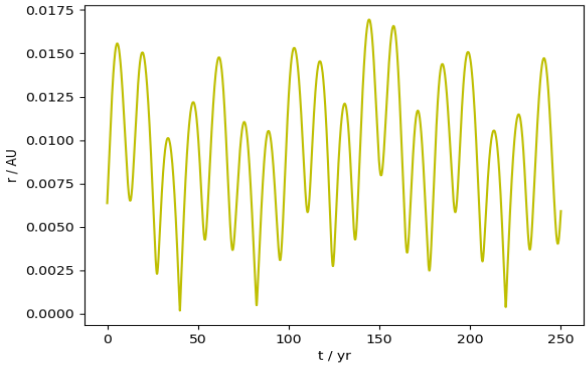
Fig. 4 shows the distances of all the planets to the center of gravity for the time span of 250 years and reflects the stability of the solution using this method.



(a) Inner planets



(b) Outer planets



(c) Sun

Figure 4: Distance to center of gravity, using `scipy.integrate.odeint` for 250 years

For the second verification, the solver `scipy.integrate.solve_ivp` was used [17]. It utilizes the explicit Runge-Kutta method of order 5(4) on default, which is based on the same idea as the Runge-Kutta-Fehlberg method, but derived differently, which results in a different butcher table and other properties, for further information see [10]. Fig. 5 and 6 show the results for 250 years using this method. We see that, when applied the solution shows great instability, since it already crashes after one year. Therefore this solver does not fit our problem. These results also indicate, that our initial value problem is a stiff one, because this method does not take stiffness into account.

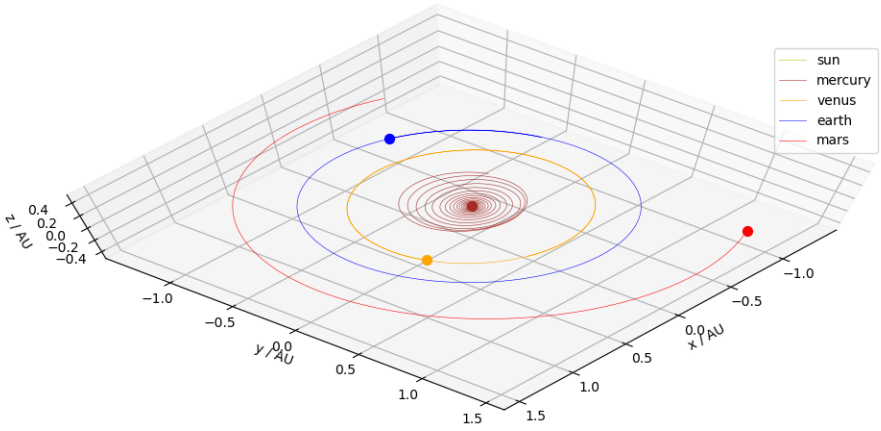


Figure 5: Orbits inner planets, using `scipy.integrate.odeint` for 250 years

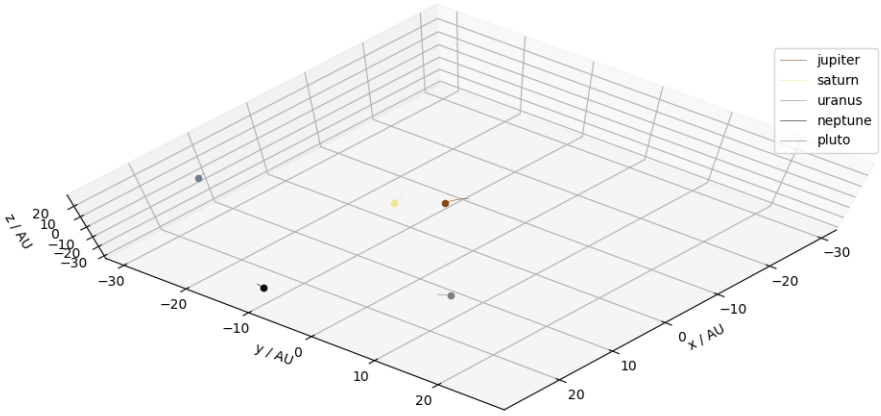
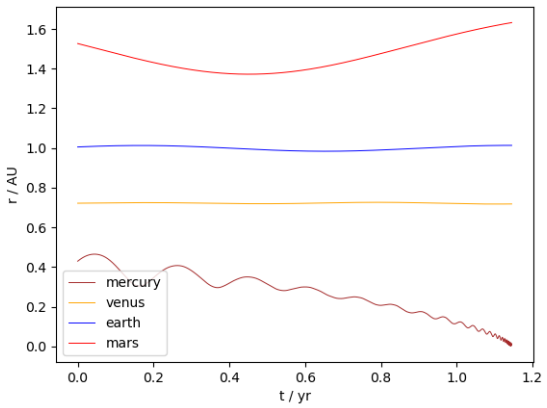
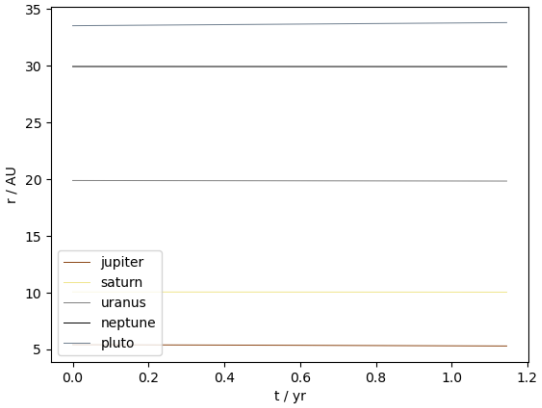


Figure 6: Orbits outer planets, using `scipy.integrate.odeint` for 250 years

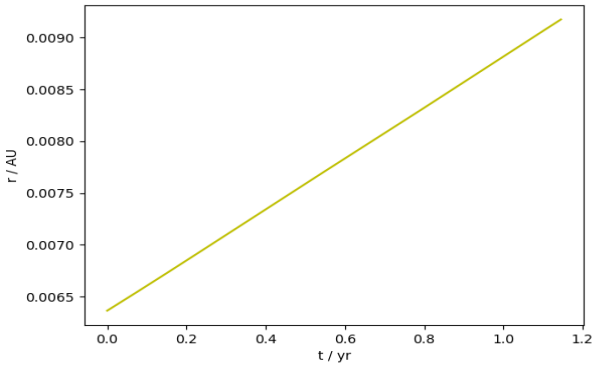
Fig. 7 shows the distances of all the planets to the center of gravity and displays the great instability of mercury.



(a) Inner planets



(b) Outer planets



(c) Sun

Figure 7: Distance to center of gravity, using `scipy.integrate.solve_ivp` for 250 years

As a result we validated our initial value problem and the next step is to implement the chosen methods from scratch.

4.2 Implementation of the Runge-Kutta method 4th order

The Runge-Kutta method 4th order was implemented as follows. A function `RK4(f,t,dt,initial)` was defined with the following inputs.
- `f` ... function describing our initial value problem A.1
- `t` ... time span starting from zero
- `dt` ... time step
- `initial` ... vector for the initial values

The size of the matrix `y` is defined through the size of the initial value vector, which describes the rows `m` and by `int(t/dt)`, which gives the columns `n`. Then the list of time steps `tlist` is created and the matrix `y` is initialized. The final steps are a for loop of the calculations for each time step, which basically is represented by the butcher tableau Eq. 34, and to save the calculated values in the matrix `y`, see line 12.

```
1   def RK4(f,t,dt,initial):
    m     = initial.shape[0]
3   n     = int(t/dt)
    tlist = np.linspace(0,t,n)
5   y     = np.zeros((m,n),float)
    y[:,0]= initial
7   for i in range(0,n-1):
    k1        = f(y[:,i])
9   k2        = f(y[:,i] + 0.5*dt*k1)
    k3        = f(y[:,i] + 0.5*dt*k2)
11  k4        = f(y[:,i] +     dt*k3)
    y[:,i+1] = y[:,i] + (dt/6.0)*(k1 + 2*k2 + 2*k3 + k4)
13
    return tlist, y
```

4.3 Implementation of the Runge-Kutta-Fehlberg method

The whole implemented function of the Runge-Kutta-Fehlberg method can be found in the appendix A.2. Since the implementation of the butcher tableau Eq.35 is nearly the same as seen in the previous chapter 4.2, here only the function inputs and the main difference, the adaptive step size control, will be discussed. The function was defined as
`RK45_scal(f , initial , t , dt_initial , n_max , tol , dt_min , dt_max)`
with the following inputs.
- `f` ... function describing our initial value problem A.1
- `initial` ... vector for the initial values
- `t` ... time span starting from zero
- `dt_initial` ... initial time step
- `n_max` ... maximum columns of the matrix `y` containing computed steps
- `tol` ... tolerance for the local error
- `dt_min` ... minimum value for the step size
- `dt_max` ... maximum value for the step size

A derivation on how one could control the step size was done in chapter 3.2.1. The way this was utilized in the implementation is shown below.

```
    if err <= tol:
2       y[:,k+1]  = y[:,k] + dt*(0.1157*k1 + 0.5489*k3  + 0.5353*k4   -   0.2*k5)
        t += dt
4       k += 1
        errlist.append(err)
6   if err == 0: err =  errlist[-1]
        dt = dt*0.84*(tol/err)**0.25
8   if dt > dt_max:
        dt = dt_max
10  elif dt < dt_min:
        dt = dt_min
```

Starting with the first if-condition, which checks if the error `err` made in the current time step is smaller than our predefined error tolerance `tol`. If that is `False`, our error `err` is bigger than the error tolerance. Therefore the choice of `dt` is rejected and we jump to line 6, which prevents zero division. Then we change the time step by multiplying it with q, see line 7. Since in this case the calculation has to be repeated, it is common to choose q conservatively and include a factor of 2 [13].

$$q = \left(\frac{tol\Delta t}{2|\tilde{y}_{n+1} - y_{n+1}|}\right)^{1/4} = 0.84\left(\frac{tol}{err}\right)^{1/4} \tag{44}$$

Now all that is left to do is to check if the new time step does not exceeds the limits `dt_min` and `dt_max`.

In the other case, where the first if-condition is `True`, `dt` is accepted and the values are computed and saved in the matrix `y`. The used time step is added to the time `t` and we go on the next step `k += 1`, see line 1-5. The other steps stay the same.

4.4 Error estimation

When using Runge-Kutta methods, we must differ between the local error made at every time step, which is typically calculated with an embedded pair of methods, like in our case with the Runge-Kutta-Fehlberg method, see Sec. 3.2, and the global error. The local error, Eq. 39, was calculated in the following way.

```
1   err = max(abs(0.00278 * k1-0.02994 * k3-0.02920 * k4+0.02000 * k5+0.03636 * k6) / dt)
```

For each step six local errors are calculated for each planet, from which the maximum will be chosen for the step size control.

There are different methods on how to estimate the global error, for further information see [14]. Here none of those methods will be used, instead the property of time reversal symmetry of our macroscopic physical system is utilized. Thus we can simply reverse our computation to get another solution, which we then can use to estimate the global error made for the time span. This can be done by changing the signs of the all planets velocities of the solution at the end of the calculated time span. Then using these values as initial condition, we run the method again for the same time span. When finished the method should have reached the original initial conditions, see Tab. 1, with a preferably small discrepancy.

5 Results and discussion

With the implementations explained above, we now can analyze the results of our simulation. To decrease the computational intensity of the simulation the dwarf planet Pluto will not be included. The outer most planet now is Neptune, therefore the simulation will be done for 165 years, which is roughly the orbital period of Neptune. To compare the solutions, the same amount of time steps that the Runge-Kutta-Fehlberg method produced were used for the Runge-Kutta method of 4th order. For simplicity the shortcut RK4 will be used for the Runge-Kutta 4th order and RKF will be used for Runge-Kutta-Fehlberg.

5.1 Runge-Kutta method 4th order

First let us look at the results of the RK4 method. Fig. 8 and 9 show the orbits of the planets. We see, with the amount of time steps from the RKF method we obtain a stable solution. This is also indicated in Fig. 10, where the distances of the planets to the center of gravity over the time span is shown.

The estimated global errors for the three most inner planets using the RK4 method described in Sec. 4.4 are shown in Fig. 11. Since Mercury represents the inner most planet it has the highest rate of change. Which means it is most sensible to step size changes. In the figures we see that mercury has the biggest global error being proportional to 10^{-4}, while the global errors of Venus and Earth are already of much smaller magnitude, 10^{-8} and 10^{-9}.

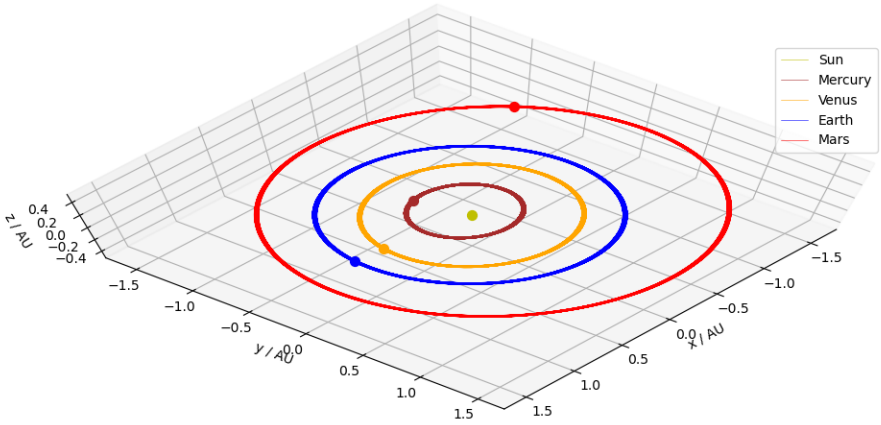


Figure 8: Orbits inner planets, using Runge-Kutta 4th order for 165 years

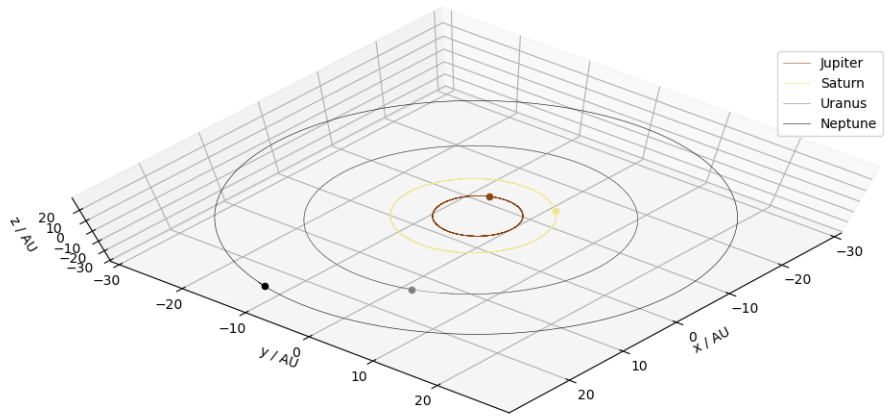
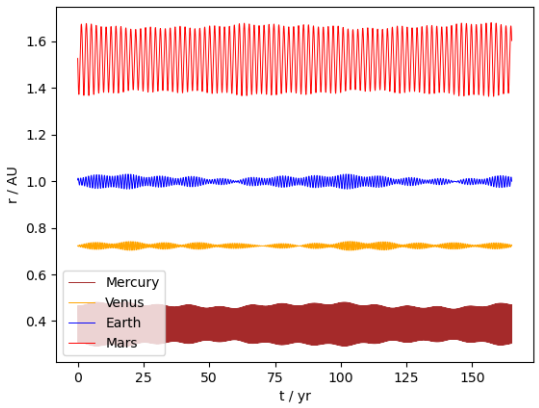
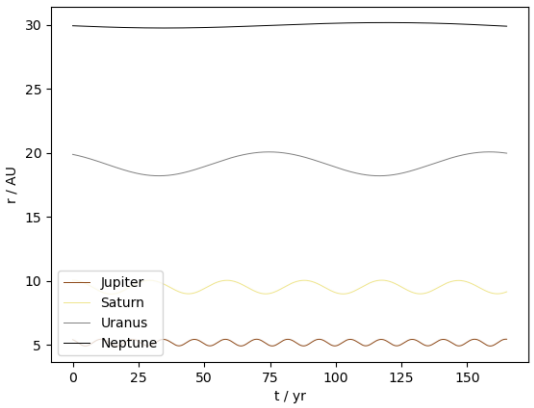


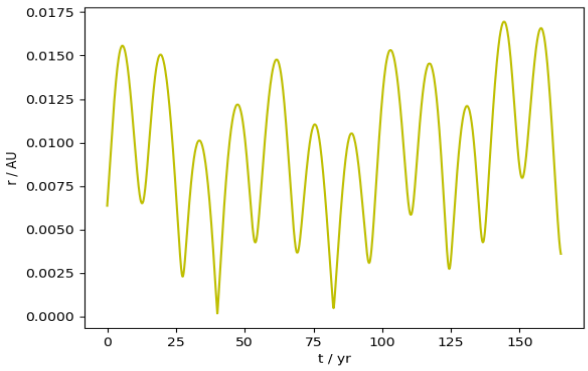
Figure 9: Orbits outer planets, using Runge-Kutta 4th order for 165 years



(a) Inner planets



(b) Outer planets



(c) Sun

Figure 10: Distance to center of gravity, using Runge-Kutta 4th order for 165 years

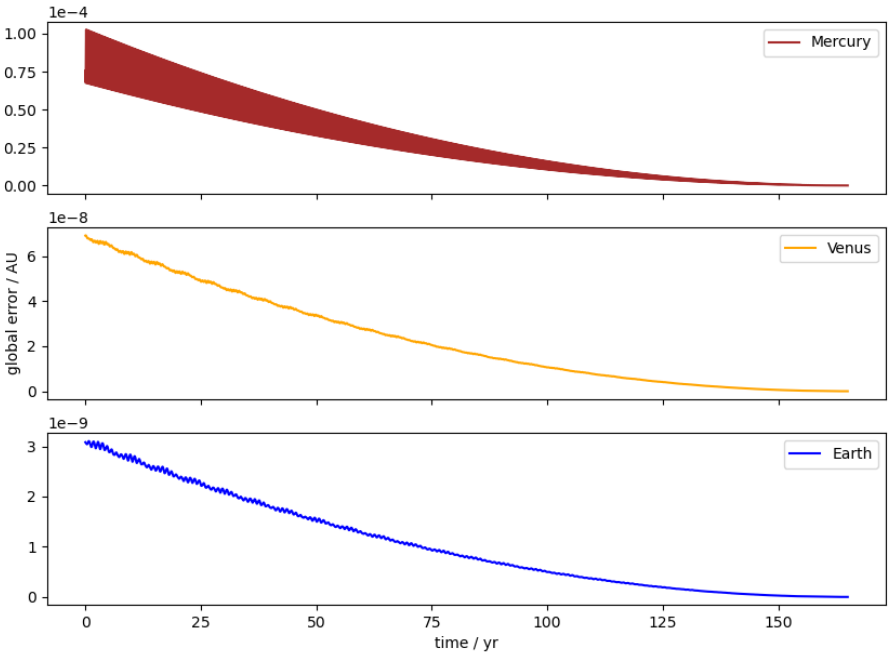


Figure 11: Global error, using Runge-Kutta 4th order for 165 years

5.2 Runge-Kutta-Fehlberg method

For the RKF method an error tolerance `tol` of 10^{-5} was chosen. Fig. 12 and 13, show the orbits calculated with the RKF method, which too created stable orbits for all the chosen planets, also indicated in Fig. 14.

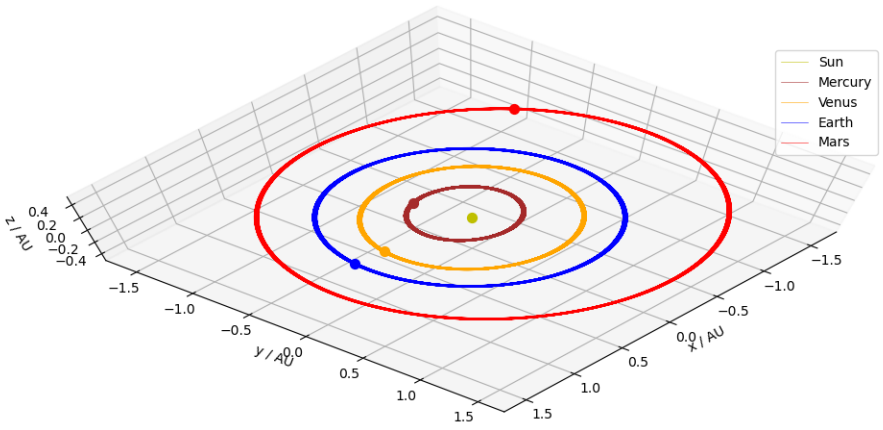


Figure 12: Orbits inner planets, using Runge-Kutta-Fehlberg method for 165 years

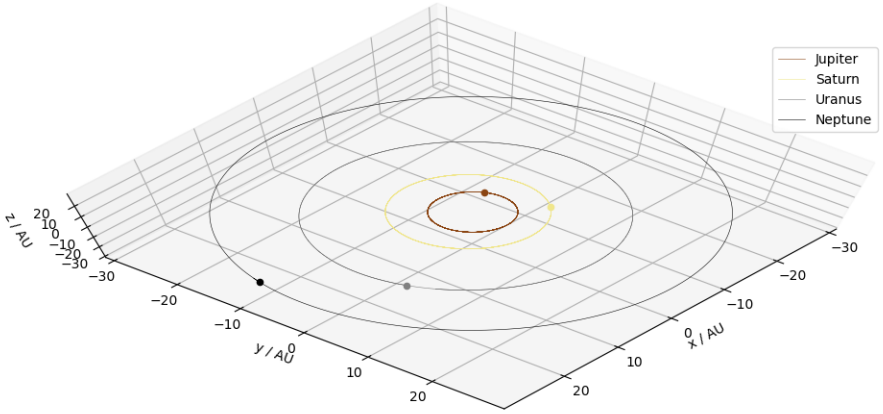
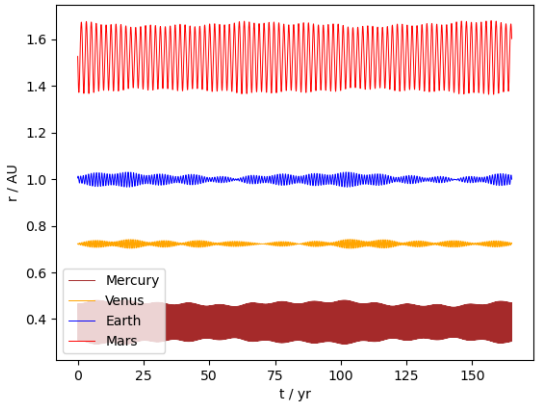
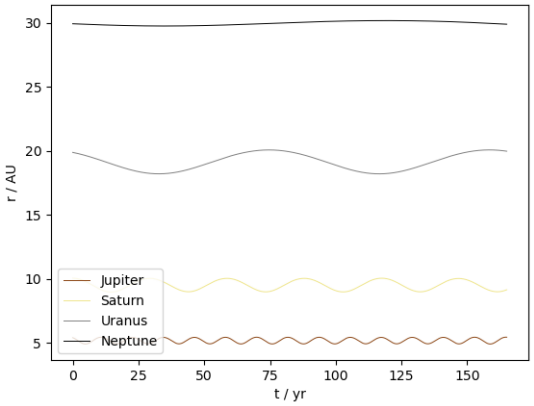


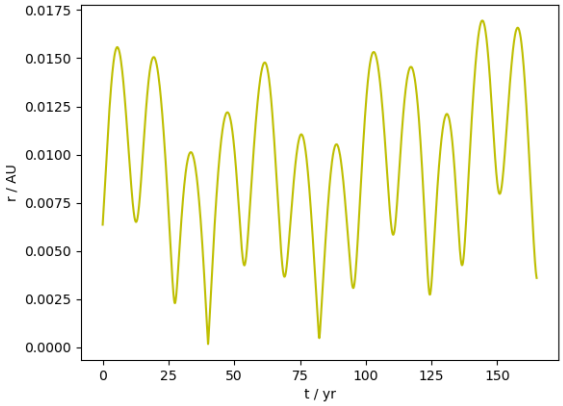
Figure 13: Orbits outer planets, using Runge-Kutta-Fehlberg method order for 165 years



(a) Inner planets



(b) Outer planets



(c) Sun

Figure 14: Distance to center of gravity, using Runge-Kutta-Fehlberg method for 165 years

When looking at the global error of the RKF method, see Fig. 15, the first difference between the methods is noticeable, because the global errors of the RKF method are one magnitude smaller than the global errors of the RK4 method.

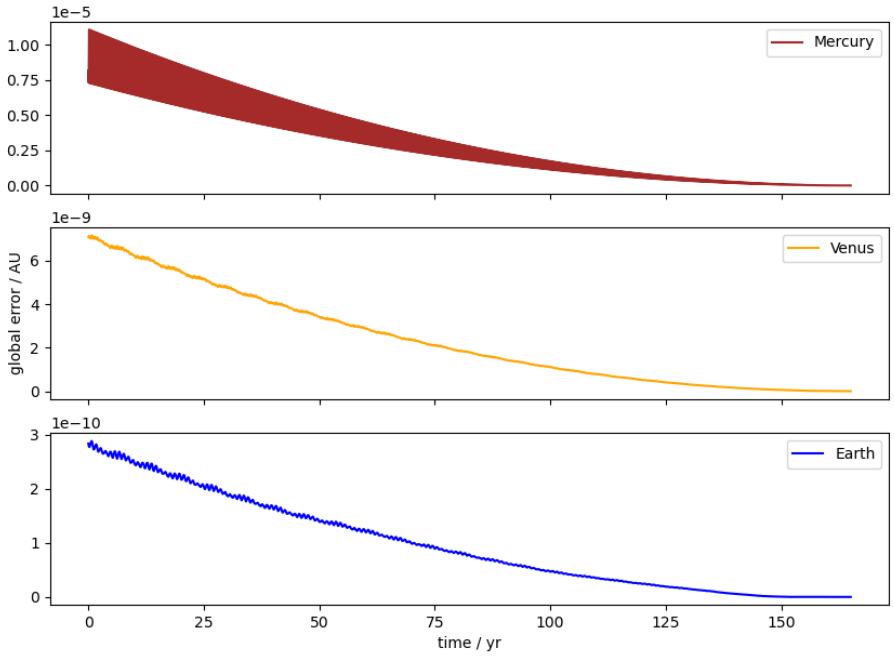


Figure 15: Global error, using Runge-Kutta-Fehlberg method for 165 years

Fig. 16 shows how our RKF method and therefore the step size reacts on the local error. The local error always arises from the coordinates or velocities of Mercury, therefore it is sufficient to look at one orbital period to analyze the pattern. We see that the error is always kept below our tolerance of 10^{-5} and the step size increases when the error decreases and vice versa. It is also noticeable, that the step size undergoes just one major change in one period of Mercury, right between the interval of $[0.15, 0.20]$ years. Reason being that the maximum local errors before and after the peak, arise from the velocities of mercury, but the few points in the interval come from the coordinates of mercury. The change in source of the local error can be explained through the fact that the time span, where this peak arises is the time, where Mercury is closest to the center of gravity and therefore has the highest velocity, see Fig. 16, also explained by Kepler's second law. As a result the change of coordinates at these points are higher than the change of velocities and the coordinates produce a bigger local error. Eventually we can say, that our designed RKF method is implemented correctly and that in case of the global error it is superior to the RK4 method.

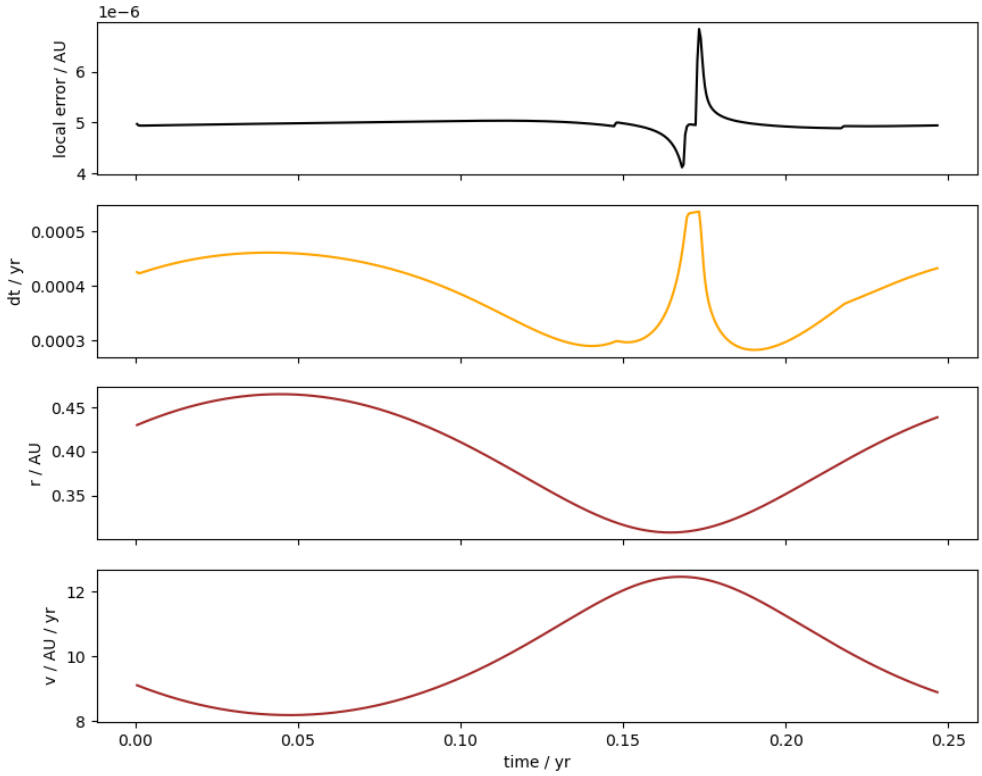


Figure 16: In descending order the figure shows the local error, the time step size, the distance to the center of gravity and the velocity of Mercury for the time span of one orbital period of Mercury, all in astronomical units. Results from the Runge-Kutta-Fehlberg method

5.3 Performance

Now we have seen that the RKF method has the advantage of a lower global error. However the trade off, that has to be made is the computational time. The implementations made are by no means perfect and there is for sure a lot of room for improvements, but since the goal is rather a functional comparison than an efficiency one, this will not be covered here. Nevertheless the performances of our methods are still of interest. Overviews of the results for the time spans 10, 80 and 165 $years$, each for the tolerances $1e-3$, $1e-4$ and $1e-5$, are shown in Tabs. 2. For timing our methods the `time` package was used. In each of the tables the same behavior can be observed. For the RKF method, when the tolerance is decreased the needed time steps and the calculation time nearly doubles, while the maximum global error shrinks one or two magnitudes. The RK4 methods behaves the same way just with lower computation times and with respect to the RKF method, a one magnitude bigger maximum global error.

time span = 10 years				
RKF	tol	1e-3	1e-4	1e-5
	t_{steps}	5803	12461	26858
	t_{comp} / min	0.4354238510131836	0.7224167784055074	1.551429855823517
	E_{gmax}/AU	6.324799326139882e-5	1.7776841022741093e-6	3.853183804063130e-8
RK4	t_{calc} / min	0.3022759437561035	0.47396289110183715	1.0209315538406372
	E_{gmax} / AU	7.687858820876831e-4	1.684447795079383e-5	3.621819116954409e-7

(a) Overview of the results for a time span of 10 years

time span = 80 years				
RKF	tol	1e-3	1e-4	1e-5
	t_{steps}	46446	99755	215036
	t_{comp} / min	2.689369543393453	5.737075754006704	12.945734874407451
	E_{gmax}/AU	4.287113386273406e-3	1.2016676188255333e-4	2.603142799379255e-6
RK4	t_{calc} / min	1.7725736896197002	3.693226182460785	8.261975761254629
	E_{gmax} / AU	5.121549056708939e-2	1.1216901076631597e-3	2.409915858624199e-5

(b) Overview of the results for a time span of 80 years

time span = 165 years				
RKF	tol	1e-3	1e-4	1e-5
	t_{steps}	95787	205742	443519
	t_{comp} / min	5.723070867856344	12.526648565133412	27.21232525507609
	E_{gmax}/AU	1.8221851939517184e-2	5.111876578374915e-4	1.1062562052375187e-5
RK4	t_{calc} / min	3.8809262911478677	7.784957464536031	17.630822185675303
	E_{gmax} / AU	2.146574141189477e-1	4.778121594349477e-3	1.0263975218689783e-4

(c) Overview of the results for a time a time of 165 years

Table 2: Overview of the results for the calculation of the Runge-Kutta-Fehlberg (RKF) method and the Runge-Kutta method 4th order (RK4). Displayed for three different time spans and three different error tolerances (tol) for the RKF step size control.
t_{steps} ... time steps calculated of the RKF method
t_{comp} ... computational time of the method in minutes
E_{gmax} ... maximum global error of the method in astronomical length

6 Summary

Through Newton's law of gravity, an initial value problem for our solar system was set up. With the initial values Tab. 1, this problem can be solved. First the initial value problem was implemented and validated with built in Runge-Kutta methods. Then a Runge-Kutta method 4th order and a Runge-Kutta-Fehlberg method was implemented from scratch. Applied on our problem the advantages and disadvantages of our implemented methods were analyzed. It was found, that both methods form stable solutions. In conclusion, with the adaptive step size control of the Runge-Kutta-Fehlberg method, we were able to achieve smaller global errors, at the cost of computational time, with respect to the Runge-Kutta method 4th order.

List of Figures

1 Illustration of the Solar System [6] 3
2 Orbits inner planets, using `scipy.integrate.odeint` for 250 years 12
3 Orbits outer planets, using `scipy.integrate.odeint` for 250 years 12
4 Distance to center of gravity, using `scipy.integrate.odeint` for 250 years 13
5 Orbits inner planets, using `scipy.integrate.odeint` for 250 years 14
6 Orbits outer planets, using `scipy.integrate.odeint` for 250 years 14
7 Distance to center of gravity, using `scipy.integrate.solve_ivp` for 250 years . . . 15
8 Orbits inner planets, using Runge-Kutta 4th order for 165 years 18
9 Orbits outer planets, using Runge-Kutta 4th order for 165 years 19
10 Distance to center of gravity, using Runge-Kutta 4th order for 165 years 19
11 Global error, using Runge-Kutta 4th order for 165 years 20
12 Orbits inner planets, using Runge-Kutta-Fehlberg method for 165 years 21
13 Orbits outer planets, using Runge-Kutta-Fehlberg method order for 165 years 21
14 Distance to center of gravity, using Runge-Kutta-Fehlberg method for 165 years . . . 22
15 Global error, using Runge-Kutta-Fehlberg method for 165 years 23
16 In descending order the figure shows the local error, the time step size, the distance to the center of gravity and the velocity of Mercury for the time span of one orbital period of Mercury, all in astronomical units. Results from the Runge-Kutta-Fehlberg method 24

References

[1] Alan B. Chamberlin. Jpl horizons on-line ephemeris system. `https://ssd.jpl.nasa.gov/horizons.cgi`. Accessed: 2020-08-13.

[2] Carl D. Murray an Stanley F. Dermott. *Solar System Dynamics*. the press syndicate of the university of cambridge, 1 edition, 1999.

[3] Ewald Schachinger Benjamin A. Stickler. *Basic Concepts in Computational Physics*. Springer, 2 edition, 2016.

[4] Dr. Lothar Collatz. *The Numerical Treatment Of Differential Equations*. Springer Verlag, 3 edition, 1960.

[5] Erwin Fehlberg. *Low-order Classical Runge-Kutta Formulas with Stepsize Control and Their Application to some Heat Transfer Problems*. National Aeronautics and Space Administration, 1969.

[6] Figure:. Solar system illustration. `https://taylorsciencegeeks.weebly.com/uploads/5/9/2/0/59201005/127261592.jpg`. Accessed: 2020-08-30.

[7] Thorsten Fließbach. *Mechanik, Lehrbuch zur Theoretischen Physik I*. Springer Sprektrum, 7 edition, 2015.

[8] Torsten Fließbach. *Allgemeine Relativitätstheorie*. Springer Sprektrum, 6 edition, 2012.

[9] Arnold Hanslmeier. *Einführung in Astronomie und Astrophysik*. Springer Sprektrum, 3 edition, 2014.

[10] P. J. Prince J. R. Dormand. *A family of embedded Runge-Kutta formulae*. Journal of Computational and Applied Mathematics, Vol. 6, No. 1, pp. 19-26, 1980.

[11] Jonathan Njeunje and Dinuka Sewwandi de Silva. the equations of planetary motion and their numerical solution. `http://www.wiu.edu/cas/mathematics_and_philosophy/graduate/`. Accessed: 2020-08-14.

[12] Assoz.-Prof. Peter Puschnig. *Computerorientierte Physik*. Lecture notes, Karl-Franzens-Universität Graz, 2019.

[13] Annete M. Burden Richard L. Burden, Douglas J. Faires. *Numerical Analysis*. CENGAGE Learning, 10 edition, 2016.

[14] Robert D. Skeel. *Thirteen Ways to Estimate Global Error*. University of Illinois at Urbana-Champaign, Department of Computer Science, 1986.

[15] Solar System Exploration Public Engagement teams. Our Solar System in depth. `https://solarsystem.nasa.gov/solar-system/our-solar-system/in-depth/`. Accessed: 2020-08-30.

[16] The SciPy community. SciPy method odeint. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html`. Accessed: 26.10.2020.

[17] The SciPy community. SciPy method solve˙ivp. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html`. Accessed: 26.10.2020.

[18] Torrence V.Johnson Tilman Spohn, Doris Breuer. *Encyclopedia of the Solar System.* Elsevier, 3 edition, 2014.

[19] Frank Hettlich Christian Karpfinger Hellmuth Stachel Tilo Arens, Rolf Busam. *Grundwissen Mathematikstudium.* Springer Sprektrum, 1 edition, 2013.

[20] Greg von Winckel. *Numerical Methods For Differential Equation.* Lecture notes, Karl-Franzens-Universität Graz, 2012.

A Python code

A.1 Initial value problem

Implementation of the 2N coupled first order differential equations 16:

```
def IVP_SS_ty(t,y):
Np   = 9
mAU = 1.98854e+30
G    = 4*np.pi**2

# planet masses
mj  = np.array([1.98854e+30, 3.30200e+23, 4.86850e+24, 5.97219e+24, 6.41850e+23,
1.89813e+27, 5.68319e+26, 8.683119e+26, 1.02410e+26, 1.30700e+22])/mAU

dy = np.zeros([Np*6,],dtype=float)

# indices for y
xi  = np.arange(0,Np*6,6)
yi  = np.arange(1,Np*6,6)
zi  = np.arange(2,Np*6,6)
vxi = np.arange(3,Np*6,6)
vyi = np.arange(4,Np*6,6)
vzi = np.arange(5,Np*6,6)

# indices for dy
dvxi = xi
dvyi = yi
dvzi = zi
daxi = vxi
dayi = vyi
dazi = vzi

# assign velocities
dy[dvxi] = y[vxi]
dy[dvyi] = y[vyi]
dy[dvzi] = y[vzi]

ax = np.zeros(Np,)
ay = np.zeros(Np,)
az = np.zeros(Np,)

for a in range(Np):
     Gm = G*np.delete(mj, a, 0)

     xa = y[xi][a]
     ya = y[yi][a]
     za = y[zi][a]

     xlist = np.delete(y[xi] - xa, a, 0)
     ylist = np.delete(y[yi] - ya, a, 0)
     zlist = np.delete(y[zi] - za, a, 0)

     rja3 = np.sqrt(np.square(xlist) + np.square(ylist) + np.square(zlist))**3

     ax[a] = np.sum(Gm*xlist/rja3)
     ay[a] = np.sum(Gm*ylist/rja3)
     az[a] = np.sum(Gm*zlist/rja3)
dy[daxi] = ax
dy[dayi] = ay
dy[dazi] = az

return dy
```

A.2 Implemented Runge-Kutta-Fehlberg method

```
    def RK45_scal(f , initial , t , dt_initial , n_max , tol , dt_min , dt_max):

    m     = initial.shape[0]
    t     = 0
    dt    = dt_initial
    k     = 0

    dtlist    = []
    tlist     = []
    tlist.append(t)
    errlist = []

    y4        = np.zeros((m,n_max),float)
    y4[:,0] = initial
    while k < n_max and t < t_final:
        # check dt for limitations
        if t+dt > t_final:
            dt = t_final - t
        # butcher tableau
        k1          = f(y4[:,k])
        k2          = f(y4[:,k] + 0.25*dt*k1)
        k3          = f(y4[:,k] + dt*(0.09375*k1 + 0.28125*k2))
        k4          = f(y4[:,k] + dt*( 0.8793809740555303*k1 - 3.277196176604461*k2 +
    3.3208921256258535*k3))
        k5          = f(y4[:,k] + dt*( 2.0324074074074074*k1 - 8.0*k2 + 7.173489278752436*
    k3 - 0.20589668615984405*k4))
        k6          = f(y4[:,k] + dt*( -0.2962962962962963*k1 + 2.0*k2 -
    1.3816764132553607*k3 + 0.4529727095516569*k4 - 0.275*k5))

        err = max(abs( 0.002777777777777778 * k1 - 0.02994152046783626 * k3 -
    0.029199893673577886 * k4 + 0.02 * k5 + 0.03636363636363636 * k6 ) / dt)
        # step size control
        if err <= tol:
            t += dt
            y4[:,k+1]  = y4[:,k] + dt*(0.11574074074074074*k1 + 0.5489278752436647*k3  +
    0.5353313840155945*k4   - 0.2*k5)
            dtlist.append(dt)
            tlist.append(t)
            errlist.append(err)
            k += 1
        if err == 0: err =  errlist[-1] # exclude zero division
            dt = dt*0.84*(tol/err)**0.25
        if dt > dt_max:
            dt = dt_max
        elif dt < dt_min:
            dt = dt_min

    return errlist , dtlist , tlist , y4
```