Project 3 FYS4150
Ordinary differential equations

Audun Tahina Reitan and Marius Holm

October 22, 2018

1 Abstract

We give a brief introduction to differential equations and how we can discretize such equations in order to solve them using computer algorithms.

More of our results can be found in the GitHub repository linked in the implementation section.

2 Introduction

In this project we'll develop code for simulating the solar system, using the Verlet algorithm for solving coupled ordinary differential equations. For this project we will focus on the use of classes in our code, which makes it a lot easier to reuse larger parts of our code for problems of similar character. In our case of the solar system we can create a class describing a planet, a moon, or some other astronomical body which we want to include in our solar system.

In order to test that our algorithm works, we start by looking at a simple hypothetical system consisting of only the Earth orbiting the Sun. We then investigate the stability of our algorithm and plot the position of the Earth orbiting the Sun. Furthermore we consider the initial velocity needed for the planet to escape the gravity of the Sun. Then we'll consider a three-body system consisting of the Earth, the Sun, and Jupiter. We then expand our three-body model to the entire solar system including Pluto. In the final part of our project we look at the perihelion precession of Mercury.

3 Methods

3.1 Differential equations

The order of an ordinary differential equation (ODE) is given by the highest order of derivative found in the left-hand side of the equation. A first order differential equation is typically of the form

$$\frac{dy}{dt} = f\big(t, \frac{dy}{dt}, y\big) \tag{1}$$

where f is an arbitrary function. A well known second order differential equation is Newton's second law

$$m\frac{d^2x}{dt^2} = -kx \tag{2}$$

where k is the force constant. All ODE's depend on one variable only, compared to partial differential equations which can depend on several variables. PDE's are widely used, but won't be of any concern in this project.

3.2 Newton's law of gravitation

The first part of our project we take a look at a simplified solar system consisting of the Sun and the Earth. Newton's law of gravitation then takes the following form:

$$F_G = \frac{GM_\odot M_{Earth}}{r^2}, \tag{3}$$

where $M_\odot$ is the mass of the Sun and M_{Earth} is the mass of the Earth. G is the gravitational constant and r is the distance between the Earth and the Sun.

We assume that the orbit of the Earth around the sun is co-planar, and we let this be the xy-plane. We can then use Newton's second law of motion which gives us two differential equations, one for each coordinate.

$$\frac{d^2x}{dt^2} = \frac{F_{G,x}}{M_{Earth}}, \tag{4}$$

and

$$\frac{d^2y}{dt^2} = \frac{F_{G,y}}{M_{Earth}}, \tag{5}$$

where $F_{G,x}$ and $F_{G,y}$ are the x and y components of the gravitational force.

3.3 Useful constants

During our project we will use astronomical units (AU) as a measure of length. 1 AU is defined as the average distance between the Sun and the Earth, that is 1 AU $= 1.5 \times 10^{11}$ m. The mass of the Sun is $M_{\text{sun}} = M_\odot = 2 \times 10^{30}$ kg. We will also need the mass of the planets we include in our code, which are given in the table below.

Planet	Mass in kg	Distance to sun in AU
Earth	$M_{\text{Earth}} = 6 \times 10^{24}$ kg	1 AU
Jupiter	$M_{\text{Jupiter}} = 1.9 \times 10^{27}$ kg	5.20 AU
Mars	$M_{\text{Mars}} = 6.6 \times 10^{23}$ kg	1.52 AU
Venus	$M_{\text{Venus}} = 4.9 \times 10^{24}$ kg	0.72 AU
Saturn	$M_{\text{Saturn}} = 5.5 \times 10^{26}$ kg	9.54 AU
Mercury	$M_{\text{Mercury}} = 3.3 \times 10^{23}$ kg	0.39 AU
Uranus	$M_{\text{Uranus}} = 8.8 \times 10^{25}$ kg	19.19 AU
Neptun	$M_{\text{Neptun}} = 1.03 \times 10^{26}$ kg	30.06 AU
Pluto	$M_{\text{Pluto}} = 1.31 \times 10^{22}$ kg	39.53 AU

Table 1: Mass and distance from the Sun for each planet in our solar system.

3.4 Discretizing differential equations

3.5 Implementation

.[Hjort-Jensen, 2015]

All our code, calculations, and plots used can be found in Auduns GitHub repository.

4 Results

4.1 Preservation of dot product

We want to show that an orthogonal transformation preserves the dot product and orthogonality. Given a basis of vectors $\mathbf{v}_i$

$$\mathbf{v}_i = \begin{bmatrix} v_{i1} \\ v_{i2} \\ \vdots \\ v_{in} \end{bmatrix} \tag{6}$$

And assuming that the basis is orthogonal.

$$\mathbf{v}_j^T \mathbf{v}_i = \delta_{ij} \tag{7}$$

The transformation is given by

$$\mathbf{w}_i = \mathbf{U}\mathbf{v}_i \tag{8}$$

As the transformation matrix $\mathbf{U}$ is orthogonal we have $\mathbf{U}^T\mathbf{U} = \mathbf{I}$.

$$\begin{aligned} \mathbf{w}_i \cdot \mathbf{w}_j = \mathbf{U}\mathbf{v}_i \cdot \mathbf{U}\mathbf{v}_j = (\mathbf{U}\mathbf{v}_i)^T(\mathbf{U}\mathbf{v}_j) = (\mathbf{U}^T\mathbf{v}_i^T)(\mathbf{U}\mathbf{v}_j) \\ = \mathbf{v}_i^T(\mathbf{U}^T\mathbf{U})\mathbf{v}_j = \mathbf{v}_i^T\,\mathbf{I}\,\mathbf{v}_j = \mathbf{v}_i^T\,\mathbf{v}_j = \mathbf{v}_i \cdot \mathbf{v}_j = \mathbf{v}_i^T\mathbf{v}_j \end{aligned}$$

As $\mathbf{w}_i \cdot \mathbf{w}_j = \mathbf{v}_i \cdot \mathbf{v}_j$ the dot product is preserved. The same applies for $\mathbf{v}_i \cdot \mathbf{v}_j = \mathbf{v}_i^T \mathbf{v}_j = \delta_{ij}$ which implies that orthogonality is also preserved. This means that the Jacobi's rotational method which multiple orthogonal transfromations preserves the orthogonality and dot product of the columns in the system we transform.

4.2 Eigenvalues of one electron in the harmonic oscillator

Using one electron in the harmonic oscillator and solving using the Jacobi's method we get the following result on the apporixmation of the analytical eigenvalues $\lambda = 3, 7, 11, 15$ for the system, varying the number of integration points N:

	N						
Absolute error	10	25	50	75	100	150	200
$\lvert\lambda_1 - \hat{\lambda}_1\rvert$	8.05E−2	1.25E−2	3.13E−3	1.39E−3	7.81E−4	3.47E−4	1.95E−4
$\lvert\lambda_2 - \hat{\lambda}_2\rvert$	4.16E−2	6.30E−2	1.57E−2	6.95E−3	3.91E−3	1.73E−3	9.74E−4
$\lvert\lambda_3 - \hat{\lambda}_3\rvert$	1.06	1.54E−1	3.81E−2	1.68E−2	9.34E−3	4.04E−3	2.18E−3
$\lvert\lambda_4 - \hat{\lambda}_4\rvert$	2.08	2.83E−1	6.54E−2	2.57E−2	1.18E−2	2.02E−3	1.42E−3

Table 2: Absolute error between the analytically known eigenvalues of the one-electron energies and our numerically calculated eigenvalues.

4.3 Quantum dots, plots

5 Discussion

5.1 Jacobi method

The Jacobi method is generally a slower algorithm than those based tridiagonalization. However the Jacobi method is easy to parallelize which can improve performance significantly. The main reason for the Jacobi methods slow convergence is that for each new rotation, matrix elements which were zero might change to non-zero values.

For the case with one electron we would have needed to choose $N > 200$ in order to get numerical eigenvalues, $\hat{\lambda}_i$ that are able to reproduce the analytical ones, λ_i with four leading digits after the decimal point (which means an absolute error $< 10^{-4}$). The Jacobi method is very slow for too large N values, and doing the calculations with an $N > 200$ a modern laptop takes several minutes to complete the calculations.

It would be of interest to investigate more effective methods like the Householder's method for large values of N, and see what differences a significant increase in integration points would result in.

References

[Hjort-Jensen, 2015] Hjort-Jensen, M. (2015). Computational physics. lecture notes. Accessible at course github repository. 551 pages.