Ordinary differential equations

Morten Hjorth-Jensen Email morten.hjorth-jensen@fys.uio.no[1,2]

Department of Physics and Center of Mathematics for Applications, University of Oslo[1]

National Superconducting Cyclotron Laboratory, Michigan State University[2]

Fall 2015

Differential equations program

- Ordinary differential equations, Runge-Kutta method,chapter 8
- Ordinary differential equations with boundary conditions: one-variable equations to be solved by shooting and Green's function methods, chapter 9
- We can solve such equations by a finite difference scheme as well, turning the equation into an eigenvalue problem. Still one variable. Done in projects 1 and 2.
- If we have more than one variable, we need to solve partial differential equations, see Chapter 10

The material on differential equations is covered by chapters 8, 9 and 10. One of the final projects, project 5, deals with ordinary differential equations.

Differential Equations, chapter 8

The order of the ODE refers to the order of the derivative on the left-hand side in the equation

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

This equation is of first order and f is an arbitrary function. A second-order equation goes typically like

$$\frac{d^2y}{dt^2} = f(t, \frac{dy}{dt}, y). \tag{2}$$

A well-known second-order equation is Newton's second law

$$m\frac{d^2x}{dt^2} = -kx, \tag{3}$$

where k is the force constant. ODE depend only on one variable

Differential Equations

partial differential equations like the time-dependent Schrödinger equation

$$i\hbar\frac{\partial\psi(\mathbf{x},t)}{\partial t} = -\frac{\hbar^2}{2m}\left(\frac{\partial^2\psi(\mathbf{r},t)}{\partial x^2} + \frac{\partial^2\psi(\mathbf{r},t)}{\partial y^2} + \frac{\partial^2\psi(\mathbf{r},t)}{\partial z^2}\right) + V(\mathbf{x})\psi(\mathbf{x},t), \tag{4}$$

may depend on several variables. In certain cases, like the above equation, the wave function can be factorized in functions of the separate variables, so that the Schroedinger equation can be rewritten in terms of sets of ordinary differential equations. These equations are discussed in chapter 10. Involve boundary conditions in addition to initial conditions.

Differential Equations

We distinguish also between linear and non-linear differential equation where for example

$$\frac{dy}{dt} = g^3(t)y(t), \tag{5}$$

is an example of a linear equation, while

$$\frac{dy}{dt} = g^3(t)y(t) - g(t)y^2(t), \tag{6}$$

is a non-linear ODE.

Differential Equations

Another concept which dictates the numerical method chosen for solving an ODE, is that of initial and boundary conditions. To give an example, if we study white dwarf stars or neutron stars we will need to solve two coupled first-order differential equations, one for the total mass m and one for the pressure P as functions of ρ

$$\frac{dm}{dr} = 4\pi r^2\rho(r)/c^2,$$

and

$$\frac{dP}{dr} = -\frac{Gm(r)}{r^2}\rho(r)/c^2.$$

where ρ is the mass-energy density. The initial conditions are dictated by the mass being zero at the center of the star, i.e., when $r = 0$, yielding $m(r = 0) = 0$. The other condition is that the pressure vanishes at the surface of the star.
In the solution of the Schroedinger equation for a particle in a potential, we may need to apply boundary conditions as well, such as demanding continuity of the wave function and its derivative.

Differential Equations

In many cases it is possible to rewrite a second-order differential equation in terms of two first-order differential equations. Consider again the case of Newton's second law in Eq. (3). If we define the position $x(t) = y^{(1)}(t)$ and the velocity $v(t) = y^{(2)}(t)$ as its derivative

$$\frac{dy^{(1)}(t)}{dt} = \frac{dx(t)}{dt} = y^{(2)}(t), \tag{7}$$

we can rewrite Newton's second law as two coupled first-order differential equations

$$m\frac{dy^{(2)}(t)}{dt} = -kx(t) = -ky^{(1)}(t), \tag{8}$$

and

$$\frac{dy^{(1)}(t)}{dt} = y^{(2)}(t). \tag{9}$$

Differential Equations, Finite Difference

These methods fall under the general class of one-step methods. The algoritm is rather simple. Suppose we have an initial value for the function $y(t)$ given by

$$y_0 = y(t = t_0). \tag{10}$$

We are interested in solving a differential equation in a region in space $[a, b]$. We define a step h by splitting the interval in N sub intervals, so that we have

$$h = \frac{b-a}{N}. \tag{11}$$

With this step and the derivative of y we can construct the next value of the function y at

$$y_1 = y(t_1 = t_0 + h), \tag{12}$$

and so forth.

Differential Equations

If the function is rather well-behaved in the domain $[a, b]$, we can use a fixed step size. If not, adaptive steps may be needed. Here we concentrate on fixed-step methods only. Let us try to generalize the above procedure by writing the step y_{i+1} in terms of the previous step y_i

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h\Delta(t_i, y_i(t_i)) + O(h^{p+1}), \tag{13}$$

where $O(h^{p+1})$ represents the truncation error. To determine Δ, we Taylor expand our function y

$$y_{i+1} = y(t = t_i+h) = y(t_i)+h(y'(t_i)+\dots+y^{(p)}(t_i)\frac{h^{p-1}}{p!})+O(h^{p+1}), \tag{14}$$

where we will associate the derivatives in the parenthesis with

$$\Delta(t_i, y_i(t_i)) = (y'(t_i) + \dots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}). \tag{15}$$

Differential Equations

We define

$$y'(t_i) = f(t_i, y_i) \tag{16}$$

and if we truncate Δ at the first derivative, we have

$$y_{i+1} = y(t_i) + hf(t_i, y_i) + O(h^2), \tag{17}$$

which when complemented with $t_{i+1} = t_i + h$ forms the algorithm for the well-known Euler method. Note that at every step we make an approximation error of the order of $O(h^2)$, however the total error is the sum over all steps $N = (b-a)/h$, yielding thus a global error which goes like $NO(h^2) \approx O(h)$.

Differential Equations

To make Euler's method more precise we can obviously decrease h (increase N). However, if we are computing the derivative f numerically by for example the two-steps formula

$$f'_{2c}(x) = \frac{f(x+h) - f(x)}{h} + O(h),$$

we can enter into roundoff error problems when we subtract two almost equal numbers $f(x+h) - f(x) \approx 0$. Euler's method is not recommended for precision calculation, although it is handy to use in order to get a first view on how a solution may look like. As an example, consider Newton's equation rewritten in Eqs. (8) and (9). We define $y_0 = y^{(1)}(t = 0)$ an $v_0 = y^{(2)}(t = 0)$. The first steps in Newton's equations are then

$$y_1^{(1)} = y_0 + hv_0 + O(h^2) \tag{18}$$

and

$$y_1^{(2)} = v_0 - hy_0k/m + O(h^2). \tag{19}$$

Differential Equations

The Euler method is asymmetric in time, since it uses information about the derivative at the beginning of the time interval. This means that we evaluate the position at $y_1^{(1)}$ using the velocity at $y_0^{(2)} = v_0$. A simple variation is to determine $y_{n+1}^{(1)}$ using the velocity at $y_{n+1}^{(2)}$, that is (in a slightly more generalized form)

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1}^{(2)} + O(h^2) \tag{20}$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2). \tag{21}$$

The acceleration a_n is a function of $a_n(y_n^{(1)}, y_n^{(2)}, t)$ and needs to be evaluated as well. This is the Euler-Cromer method.

Differential Equations

Let us then include the second derivative in our Taylor expansion. We have then

$$\Delta(t_i, y_i(t_i)) = f(t_i) + \frac{h}{2}\frac{df(t_i, y_i)}{dt} + O(h^3). \tag{22}$$

The second derivative can be rewritten as

$$y'' = f' = \frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f \tag{23}$$

and we can rewrite Eq. (14) as

$$y_{i+1} = y(t = t_i + h) = y(t_i) + hf(t_i) + \frac{h^2}{2}\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f\right) + O(h^3), \tag{24}$$

which has a local approximation error $O(h^3)$ and a global error $O(h^2)$.

Differential Equations

These approximations can be generalized by using the derivative f to arbitrary order so that we have

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h(f(t_i, y_i) + \dots f^{(p-1)}(t_i, y_i)\frac{h^{p-1}}{p!}) + O(h^{p+1}) \tag{25}$$

These methods, based on higher-order derivatives, are in general not used in numerical computation, since they rely on evaluating derivatives several times. Unless one has analytical expressions for these, the risk of roundoff errors is large.

Differential Equations

The most obvious improvements to Euler's and Euler-Cromer's algorithms, avoiding in addition the need for computing a second derivative, is the so-called midpoint method. We have then

$$y_{n+1}^{(1)} = y_n^{(1)} + \frac{h}{2}\left(y_{n+1}^{(2)} + y_n^{(2)}\right) + O(h^2) \tag{26}$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2), \tag{27}$$

yielding

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_n^{(2)} + \frac{h^2}{2}a_n + O(h^3) \tag{28}$$

implying that the local truncation error in the position is now $O(h^3)$, whereas Euler's or Euler-Cromer's methods have a local error of $O(h^2)$.

Differential Equations

Thus, the midpoint method yields a global error with second-order accuracy for the position and first-order accuracy for the velocity. However, although these methods yield exact results for constant accelerations, the error increases in general with each time step.
One method that avoids this is the so-called half-step method.
Here we define

$$y_{n+1/2}^{(2)} = y_{n-1/2}^{(2)} + ha_n + O(h^2), \tag{29}$$

and

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1/2}^{(2)} + O(h^2). \tag{30}$$

Note that this method needs the calculation of $y_{1/2}^{(2)}$. This is done using e.g., Euler's method

$$y_{1/2}^{(2)} = y_0^{(2)} + ha_0 + O(h^2). \tag{31}$$

As this method is numerically stable, it is often used instead of Euler's method.

Differential Equations

Another method which one may encounter is the Euler-Richardson method with

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_{n+1/2} + O(h^2), \tag{32}$$

and

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1/2}^{(2)} + O(h^2). \tag{33}$$

The program program2.cpp includes all of the above methods.

Differential Equations, Runge-Kutta methods

Runge-Kutta (RK) methods are based on Taylor expansion formulae, but yield in general better algorithms for solutions of an ODE. The basic philosophy is that it provides an intermediate step in the computation of y_{i+1}.
To see this, consider first the following definitions

$$\frac{dy}{dt} = f(t, y), \tag{34}$$

and

$$y(t) = \int f(t, y)dt, \tag{35}$$

and

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y)dt. \tag{36}$$

Differential Equations, Runge-Kutta methods

To demonstrate the philosophy behind RK methods, let us consider the second-order RK method, RK2. The first approximation consists in Taylor expanding $f(t,y)$ around the center of the integration interval t_i to t_{i+1}, that is, at $t_i + h/2$, h being the step. Using the midpoint formula for an integral, defining $y(t_i + h/2) = y_{i+1/2}$ and $t_i + h/2 = t_{i+1/2}$, we obtain

$$\int_{t_i}^{t_{i+1}} f(t,y)dt \approx hf(t_{i+1/2}, y_{i+1/2}) + O(h^3). \tag{37}$$

This means in turn that we have

$$y_{i+1} = y_i + hf(t_{i+1/2}, y_{i+1/2}) + O(h^3). \tag{38}$$

Differential Equations, Runge-Kutta methods

However, we do not know the value of $y_{i+1/2}$. Here comes thus the next approximation, namely, we use Euler's method to approximate $y_{i+1/2}$. We have then

$$y_{(i+1/2)} = y_i + \frac{h}{2}\frac{dy}{dt} = y(t_i) + \frac{h}{2}f(t_i, y_i). \tag{39}$$

This means that we can define the following algorithm for the second-order Runge-Kutta method, RK2.

$$k_1 = hf(t_i, y_i), \tag{40}$$

$$k_2 = hf(t_{i+1/2}, y_i + k_1/2), \tag{41}$$

with the final value

$$y_{i+i} \approx y_i + k_2 + O(h^3). \tag{42}$$

The difference between the previous one-step methods is that we now need an intermediate step in our evaluation, namely

Differential Equations, Runge-Kutta methods

The fourth-order Runge-Kutta, RK4, which we will employ in the solution of various differential equations below, has the following algorithm

$$k_1 = hf(t_i, y_i) \qquad k_2 = hf(t_i + h/2, y_i + k_1/2)$$

$$k_3 = hf(t_i + h/2, y_i + k_2/2) \qquad k_4 = hf(t_i + h, y_i + k_3)$$

with the final result

$$y_{i+1} = y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right).$$

Thus, the algorithm consists in first calculating k_1 with t_i, y_1 and f as inputs. Thereafter, we increase the step size by $h/2$ and calculate k_2, then k_3 and finally k_4. The global error goes as $O(h^4)$.

Simple Example, Block tied to a Wall

Our first example is the classical case of simple harmonic oscillations, namely a block sliding on a horizontal frictionless surface. The block is tied to a wall with a spring. If the spring is not compressed or stretched too far, the force on the block at a given position x is

$$F = -kx.$$

The negative sign means that the force acts to restore the object to an equilibrium position. Newton's equation of motion for this idealized system is then

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

or we could rephrase it as

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x = -\omega_0^2 x,$$

with the angular frequency $\omega_0^2 = k/m$.

Simple Example, Block tied to a Wall

With the position $x(t)$ and the velocity $v(t) = dx/dt$ we can reformulate Newton's equation in the following way

$$\frac{dx(t)}{dt} = v(t),$$

and

$$\frac{dv(t)}{dt} = -\omega_0^2 x(t).$$

We are now going to solve these equations using the Runge-Kutta method to fourth order discussed previously.

Simple Example, Block tied to a Wall

Before proceeding however, it is important to note that in addition to the exact solution, we have at least two further tests which can be used to check our solution.

Since functions like *cos* are periodic with a period 2π, then the solution $x(t)$ has also to be periodic. This means that

$$x(t+T) = x(t),$$

with T the period defined as

$$T = \frac{2\pi}{\omega_0} = \frac{2\pi}{\sqrt{k/m}}.$$

Observe that T depends only on k/m and not on the amplitude of the solution.

Simple Example, Block tied to a Wall

In addition to the periodicity test, the total energy has also to be conserved.
Suppose we choose the initial conditions

$$x(t=0)=1\,\mathrm{m} \qquad v(t=0)=0\,\mathrm{m/s},$$

meaning that block is at rest at $t=0$ but with a potential energy

$$E_0=\frac{1}{2}kx(t=0)^2=\frac{1}{2}k.$$

The total energy at any time t has however to be conserved, meaning that our solution has to fulfil the condition

$$E_0=\frac{1}{2}kx(t)^2+\frac{1}{2}mv(t)^2.$$

Simple Example, Block tied to a Wall

An algorithm which implements these equations is included below.

- Choose the initial position and speed, with the most common choice $v(t=0)=0$ and some fixed value for the position.
- Choose the method you wish to employ in solving the problem.
- Subdivide the time interval $[t_i,t_f]$ into a grid with step size

$$h=\frac{t_f-t_i}{N},$$

where N is the number of mesh points.

- Calculate now the total energy given by

$$E_0=\frac{1}{2}kx(t=0)^2=\frac{1}{2}k.$$

- The Runge-Kutta method is used to obtain x_{i+1} and v_{i+1} starting from the previous values x_i and v_i.
- When we have computed $x(v)_{i+1}$ we upgrade $t_{i+1}=t_i+h$.
- This iterative process continues till we reach the maximum time t_f.

Simple Example, Block tied to a Wall

To run a c++ program using ipython notebook, you can use the following statements.

```
%%install_ext https://raw.github.com/dragly/cppmagic/master/cppmagic.p
%%load_ext cppmagic

%%cpp
/*    This program solves Newton's equation for a block
      sliding on a horizontal frictionless surface. The block
      is tied  to a wall with a spring, and Newton's equation
      takes the form
           m d^2x/dt^2 =-kx
      with k the spring tension and m the mass of the block.
      The angular frequency is omega^2 = k/m and we set it equal
      1 in this example program.

      Newton's equation is rewritten as two coupled differential
      equations, one for the position x  and one for the velocity v
           dx/dt = v     and
           dv/dt = -x    when we set k/m=1

      We use therefore a two-dimensional array to represent x and v
      as functions of t
      y[0] == x
      y[1] == v
      dy[0]/dt = v
      dy[1]/dt = -x
```

The classical pendulum

The angular equation of motion of the pendulum is given by Newton's equation and with no external force it reads

$$ml\frac{d^2\theta}{dt^2}+mgsin(\theta)=0, \tag{43}$$

with an angular velocity and acceleration given by

$$v=l\frac{d\theta}{dt}, \tag{44}$$

and

$$a=l\frac{d^2\theta}{dt^2}. \tag{45}$$

More on the Pendulum

We do however expect that the motion will gradually come to an end due a viscous drag torque acting on the pendulum. In the presence of the drag, the above equation becomes

$$ml\frac{d^2\theta}{dt^2}+\nu\frac{d\theta}{dt}+mgsin(\theta)=0, \tag{46}$$

where ν is now a positive constant parameterizing the viscosity of the medium in question. In order to maintain the motion against viscosity, it is necessary to add some external driving force. We choose here a periodic driving force. The last equation becomes then

$$ml\frac{d^2\theta}{dt^2}+\nu\frac{d\theta}{dt}+mgsin(\theta)=Asin(\omega t), \tag{47}$$

with A and ω two constants representing the amplitude and the angular frequency respectively. The latter is called the driving frequency.

More on the Pendulum

We define

$$\omega_0=\sqrt{g/l},$$

the so-called natural frequency and the new dimensionless quantities

$$\hat{t}=\omega_0 t,$$

with the dimensionless driving frequency

$$\hat{\omega}=\frac{\omega}{\omega_0},$$

and introducing the quantity Q, called the *quality factor*,

$$Q=\frac{mg}{\omega_0\nu},$$

and the dimensionless amplitude

$$\hat{A}=\frac{A}{mg}$$

More on the Pendulum

We have

$$\frac{d^2\theta}{d\hat{t}^2} + \frac{1}{Q}\frac{d\theta}{d\hat{t}} + sin(\theta) = \hat{A}cos(\hat{\omega}\hat{t}).$$

This equation can in turn be recast in terms of two coupled first-order differential equations as follows

$$\frac{d\theta}{d\hat{t}} = \hat{v},$$

and

$$\frac{d\hat{v}}{d\hat{t}} = -\frac{\hat{v}}{Q} - sin(\theta) + \hat{A}cos(\hat{\omega}\hat{t}).$$

These are the equations to be solved. The factor Q represents the number of oscillations of the undriven system that must occur before its energy is significantly reduced due to the viscous drag. The amplitude $\hat{A}$ is measured in units of the maximum possible gravitational torque while $\hat{\omega}$ is the angular frequency of the external torque measured in units of the pendulum's natural frequency.

Adaptive methods

In case the function to integrate varies slowly or fast in different integration domains, adaptive methods are normally used. One strategy is always to decrease the step size. As we have seen earlier, this leads to more CPU cycles and may lead to loss or numerical precision. An alternative is to use higher-order RK methods for example. However, this leads again to more cycles, furthermore, there is no guarantee that higher-order leads to an improved error.

Adaptive methods

Assume the exact result is $\tilde{x}$ and that we are using an RKM method. Suppose we run two calculations, one with h (called x_1) and one with $h/2$ (called x_2). Then

$$\tilde{x} = x_1 + Ch^{M+1} + O(h^{M+2}),$$

and

$$\tilde{x} = x_2 + 2C(h/2)^{M+1} + O(h^{M+2}),$$

with C a constant. Note that we calculate two halves in the last equation. We get then

$$|x_1 - x_2| = Ch^{M+1}(1 - \frac{1}{2^M}).$$

yielding

$$C = \frac{|x_1 - x_2|}{(1 - 2^{-M})h^{M+1}}.$$

We rewrite

$$\tilde{x} = x_2 + \epsilon + O((h)^{M+2}),$$

Adaptive methods

With RK4 the expressions become

$$\tilde{x} = x_2 + \epsilon + O((h)^6),$$

with

$$\epsilon = \frac{|x_1 - x_2|}{15}.$$

The estimate is one order higher than the original RK4. But this method is normally rather inefficient since it requires a lot of computations. We solve typically the equation three times at each time step. However, we can compare the estimate ϵ with some by us given accuracy ξ. We can then ask the question: what is, with a given x_j and t_j, the largest possible step size $\tilde{h}$ that leads to a truncation error below ξ? We want

$$C\tilde{h} \leq \xi,$$

which leads to

$$\left(\tilde{h}\right)^{M+1} |x_1 - x_2| \leq \xi$$

Adaptive methods

With

$$\tilde{h} = h\left(\frac{\xi}{\epsilon}\right)^{1+1/M}.$$

we can design the following algorithm:

- If the two answers are close, keep the approximation to h.
- If $\epsilon > \xi$ we need to decrease the step size in the next time step.
- If $\epsilon < \xi$ we need to increase the step size in the next time step.

A much used algorithm is the so-called RKF45 which uses a combination of fourth and fifth order RK methods.

Adaptive methods, RKF45

At each step, two different approximations for the solution are made and compared. If the two answers are in close agreement, the approximation is accepted. If the two answers do not agree to a specified accuracy, the step size is reduced. If the answers agree to more significant digits than required, the step size is increased. Each step requires the use of the following six values:

$$k_1 = hf(t_k, y_k),$$

$$k_2 = hf(t_k + \frac{1}{4}h, y_k + \frac{1}{4}k_1),$$

$$k_3 = hf(t_k + \frac{3}{8}h, y_k + \frac{3}{32}k_1 + \frac{9}{32}k_2),$$

$$k_4 = hf(t_k + \frac{12}{13}h, y_k + \frac{1932}{2197}k_1 + \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3),$$

$$k_5 = hf(t_k + h, y_k + \frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 + \frac{845}{4104}k_4),$$

$$k_6 = hf(t_k + \frac{1}{}h, y_k - \frac{8}{}k_1 + 2k_2 - \frac{3544}{}k_3 + \frac{1859}{}k_4 - + \frac{11}{}k_5)$$

Adaptive methods, RKF45

An approximation to the solution of the ODE is made using a Runge-Kutta method of order 4:

$$y_{k+1} = y_k + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4101}k_4 - \frac{1}{5}k_5,$$

where the four function values k_1 , k_3 , k_4 , and k_5 are used. Notice that k_2 is not used here. A better value for the solution is determined using a Runge-Kutta method of order 5:

$$z_{k+1} = y_k + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6.$$

The optimal time step αh is then determined by

$$\alpha = \left(\frac{\xi h}{2|z_{k+1} - y_{k+1}|}\right)^{1/4},$$

with ξ our defined tolerance.

Solar system code, main program

```
#include <iostream>
#include <solarsystem.h>
#include <planet.h>
#include <cmath>
#include <armadillo>


using namespace arma;
using namespace std;



int main()
{
    solarsystem mysystem;

planet  Sun(1,0,0,0,0,0,0);
planet  Mercury(1.2e-7, 0.39, 0, 0,0,9.96,0);
planet  Venus(2.4e-6, 0.72, 0, 0,0,7.36,0);
planet  Earth(1.5e-6,1,0,0, 0, 6.26, 0);
planet  Mars(3.3e-7, 1.52, 0, 0,0,5.06,0);
planet  Jupiter(9.5e-4, 5.20, 0,0,0,2.75,0);
planet  Saturn(2.75e-4, 9.54, 0, 0,0,2.04,0);
planet  Uranus(4.4e-5, 19.19, 0, 0,0,1.43,0);
planet  Neptune(5.1e-5, 30.06, 0, 0,0,1.14,0);
planet  Pluto(5.6e-9, 39.53, 0, 0,0,0.99,0);

   mysystem.add(Sun);
```

Solar system code, planet class, header file

```
#ifndef PLANET_H
#define PLANET_H

class planet
{
public:

    double position[3];
    double velocity[3];
    double mass;

    planet(double mas, double x,double y, double z, double vx, double
    planet();

};

#endif // PLANET_H
```

Solar system code, planet class

```
#include "planet.h"

planet::planet()
{
};

planet::planet(double mas, double x,double y, double z, double vx, dou
mass = mas;
position[0] = x;
position[1] = y;
position[2] = z;

velocity[0] = vx;
velocity[1] = vy;
velocity[2] = vz;
};
```

Solar system code, solver class

```
#ifndef SOLARSYSTEM_H
#define SOLARSYSTEM_H
#include <armadillo>
#include "planet.h"
#include <vector>

using std::vector;
using namespace arma;

class solarsystem
{
public:

    int number_planets=0;

    solarsystem();
    vector<planet> all_planets;
    void add(planet n);
    void print_position(std::ofstream &output, std::ofstream &output2,
    void print_position(std::ofstream &output, std::ofstream &output2,
    void synctroniz(vector<planet> vec, arma::mat &ma);
    void insert_data(vector<planet> vec, arma::mat &ma);
    void solverRK4(vector<planet> vec, double h, double tmax);

    void solverVERLET(vector<planet> vec, double h, double tmax);
    //da main

    void sum_matrix(mat &result, double coeff_one, mat &first, double
```

Solar system code, solver code

```
#include "solarsystem.h"
#include "planet.h"
#include <iostream>
#include <armadillo>
#include <stdio.h>
#include <iomanip>
using namespace arma;
using namespace std;


solarsystem::solarsystem()
{
}
void solarsystem::add(planet n){
     number_planets++;
     all_planets.push_back(n);
 }
void solarsystem::print_position(ofstream &output, ofstream &output2,
    print_position(output, output2, vec, 3);
}

void solarsystem::print_position(ofstream &output, ofstream &output2,
    if(n>3 || n<=0) n=3;
    for(int i=0; i<vec.size(); i++){
        planet &this = vec[i];
        std::cout << std::scientific;
        for(int j=0; j<n;j++){
        std::cout << this.position[i] << "   ";
```