Computational Physics - Project 5

Johannes Scheller (candidate no. 71), Vincent Noculak (candidate no. 22)
Lukas Powalla (candidate no. 67), Richard Asbah (candidate no. 50)

December 11, 2015

Contents

Abstract

This report by Richard Absah, Vincent Noculak, Lukas Powalla and Johannes Scheller for the course "FYS3150/4150 - Computational Physics" at the Universitetet i Oslo deals with the computational simulation of a galaxy model. It contains different approaches on solving the differential equations resulting from Newton's law of universal gravitation for objects (e.g. stars) in space. The authors will introduce two algorithms, Velocity Verlet and Runge-Kutta, to solve these differential equations and show both of them in use. They will compare there advantages and disadvantages and finally justify their chose of using Velocity Verlet to simulate n-body system like galaxies. These simulations are used to obtain knowledge about the behaviour of large particle clusters in space, esp. about the collapses these structures perform. Moreover, the authors try to prove powerful statements like the virial theorem by their simulations. The paper ends with an discussion about the future use of the methods use, but also about the problems that occurred during the project. Thus, it will give a small outlook on the limits of the algorithms for future use.

1 Introduction

2 Theory

2.1 Newton's law of universal gravitation

Newton's law of universal gravitation states that every point mass attracts every single other point mass by a force pointing along the line intersecting both points. The force two points of mass with the masses m_i and m_j and a distance of r_{ij} to each other attract each other is given by

$$\mathbf{F_G} = -G\frac{m_i \cdot m_j}{r_{ij}^3} \cdot \mathbf{r_{ij}} \tag{1}$$

Where G is the gravitational constant. In our project we have a system with n stars which get approximated as points of mass. Hence, to get the gravitational force acting on one star, we just need to sum over all $\mathbf{F}_G$'s (having the form of equation (1)) that are acting on this star due to the other stars.

$$\mathbf{F}_{Gi} = \sum_{j\neq i}^{n} -G\frac{m_i \cdot m_j}{r_{ij}^3} \cdot \mathbf{r}_{ij} = -Gm_i \sum_{j\neq i}^{n} m_j \frac{\mathbf{r}_{ij}}{r_{ij}^3} \tag{2}$$

Then the acceleration of star i is given by

$$\mathbf{a} = -G\sum_{j\neq i}^{n} m_j \frac{\mathbf{r}_{ij}}{r_{ij}^3} \tag{3}$$

2.2 Energy

In the project we are going to observe the kinetic and potential energy of systems of particles, which interact via the gravitational force. The kinetic energy of a particle with mass m and velocity v is given by $E_{Kin} = \frac{1}{2}mv^2$. Because we have n particles we just need to sum over the kinetic energies of all particles to get the kinetic energy of the system.

$$E_{Kin} = \sum_{i=1}^{n} \frac{1}{2} m_i v_i^2 \tag{4}$$

The potential energy of a system is given by $\nabla E_{Pot} = -\mathbf{F}$. Hence we have $\frac{\partial}{\partial x_i} E_{Pot} = -F_{x_i}$ and can get a potential energy by trough integration:

$$E_{Pot} = -\int dx' F_x(x', y, z) \tag{5}$$

To get the potential energy of our system, we add one particle after an other from an infinite distance to the system. Each time we add a particle i we calculate its potential energy V_i (negative work that is needed to move the particle from an infinite distance to the system) using (5). We do this until we have n particles in the system. Then the potential energy of our system is given by the sum over all V_i. If we add the particle 1 there is no force yet, hence $V_1 = 0$. When adding the second particle there is a force between particle 1 and 2. Adding the particle number i, there are $i-1$ forces acting on particle i(we write the force between particle i and j, acting on i, as F_{ij}). Hence V_i can simply be calculated by

$$V_i = -\int_{x_i}^{\infty} dx \sum_{j\neq i}^{i} F_{ij_x} \tag{6}$$

And we get

$$V_{System} = \sum_{i=1}^{n} V_i = -\sum_{i=1}^{n} \int_{x_i}^{\infty} dx \sum_{j\neq i}^{i} F_{ij_x} = -\frac{1}{2} \sum_{i,j=1;i\neq j}^{n} \int_{x_i}^{\infty} dx F_{ij_x} \tag{7}$$

With (1) we know

$$\int_{x_i}^{\infty} dx F_{ij_x} = -\int_{x_i}^{\infty} dx G m_i m_j \frac{(x-x_j)}{((x-x_j)^2+(y_i-y_j)^2+(z_i-z_j)^2)^{\frac{3}{2}}} = -\left[G \frac{m_i m_j}{((x-x_j)^2+(y_i-y_j)^2+(z_i-z_j)^2)^{\frac{1}{2}}} \right]_{x_i}^{\infty} = G\frac{m_i m_j}{r_{ij}} \tag{8}$$

Setting (8) in (7) we get the potential energy of our system, which is

$$V_{System} = -\frac{1}{2} \sum_{i,j=1;i\neq j}^{n} G\frac{m_i m_j}{r_{ij}} \tag{9}$$

2.3 Open Clusters

Open clusters are groups of a few thousand stars that were formed from the same giant molecular cloud and have roughly the same age. They have yet only been found in spiral and irregular galaxies, in which active star formation is occurring. Many open clusters are unstable and have a small enough mass that the escape velocity of the system is lower than the average velocity of the constituent stars. These clusters will rapidly disperse within a few million years [1]

The open cluster we simulate in this project will have the starting conditions of a cold collapse. This means, that the particles start with little or no initial velocity. In addition to the that, the N particles, we simulate, will be randomly uniformly distributed at the starting time inside a sphere with given radius R_0. Our particles masses will be distributed by a Gaussian distribution around ten solar masses with a standard deviation of one solar mass.

2.4 Verlet algorithm

The Verlet algorithm is one method that is important to numerically solve a equation of motion having for example the form of equation (2).

Using a taylor expansion for a function $x(t)$ around the point $t_0 = t_i$ and then setting $t = t_i + h$ and $t = t_i - h$, we get the equations

$$x(t_i+h) = x_i + hx^{'}(t_i) + \frac{h^2}{2}x^{''}(t_i) + \frac{h^3}{3!}x^{'''}(t_i) + \sigma(h^4) \tag{10}$$

$$x(t_i-h) = x_i - hx^{'}(t_i) + \frac{h^2}{2}x^{''}(t_i) - \frac{h^3}{3!}x^{'''}(t_i) + \sigma(h^4) \tag{11}$$

For an easier overview it is useful to use the notation $x(t_i) = x_i$, $x(t_i \pm h) = x_{i\pm1}$ and $x^{''} = a(x,t)$. By adding up equation (10) and (11) we get the equation

$$x_{i+1} = 2x_i - x_{i-1} + a(x_i, t_i)\cdot h^2 + \sigma(h^4) \tag{12}$$

Which gives us a method to determine all x_i, if the boundary conditions are known. It can be seen that due to the x_{i-1} part in (12), the algorithm is not self-starting. Hence x_1 must be found, using an other method (for example Euler's method).

2.4.1 Velocity Verlet algorithm

The Velocity Verlet is similar to the Verlet algorithm and is also using Taylor expansions. It uses a half-step velocity $v_{i+\frac{1}{2}}$ ($v(t) = \frac{dx}{dt}$) to calculate x_{i+1}. The first step of the algorithm is to calculate $v_{\frac{1}{2}}$ with (13), using given v_0 and $a(x_0, t_0)$.

$$v_{i+\frac{1}{2}} = v_i + \frac{h}{2}a(x_i, t_i) \tag{13}$$

After this, the algorithm can be executed by simply repeating the following three steps for all i.

1.: Calculate

$$x_{i+1} = x_i + hv_{i+\frac{1}{2}} \tag{14}$$

2.: Calculate a_{i+1} using x_{i+1}

3.: Calculate

$$v_{i+\frac{3}{2}} = v_{i+\frac{1}{2}} + ha_{i+1} \tag{15}$$

Because we have to calculate $v_{\frac{1}{2}}$ first, the algorithm is not self-starting. It has to be mentioned, that it is also possible to eliminate the half-step velocities by setting (13) in (14) and (15). Then, we are using (15) to calculate v_{i+1}, by just using the factor $\frac{h}{2}$ instead of h in the formula. Doing this, we get the formulas

$$x_{i+1} = x_i + h \cdot v_i + \frac{h^2}{2} a_i(x_i, t_i) \tag{16}$$

and

$$v_{i+1} = v_i + \frac{h}{2} \cdot (a(x_i, t_i) + a(x_{i+1}, t_{i+1})) \tag{17}$$

We see, that (17) uses the Trapezoidal rule. This form of the algorithm starts with calculating equation (16) for $i = 0$ to get x_{i+1}. Knowing this value, $a(x_i, t_i)$ and $a(x_{i+1}, t_{i+1})$ can be calculated. Then it is possible to calculate v_{i+1} with equation (17). These steps can now be repeated for $i = i + 1$, allowing to get the values for all x_i and v_i.

2.5 Runge-Kutta methods

Runge-Kutta methods use the following basic equation of integration to numerically approximate Ordinary differential equations.

$$x_{i+1} = x_i + \int_{t_i}^{t_{i+1}} dt' \, v(x, t') \tag{18}$$

With $x(t_i) = x_i$, $x(t_i \pm h) = x_{i\pm1}$ and $\frac{dx}{dt} = v$. The integral can be approximated using numerically integration methods, as for example the trapezoidal rule or Simpson's rule. In Runge-Kutta methods to calculate a step x_{i+1}, an intermediate step between x_i and x_{i+1} gets used.

2.5.1 fourth order Runge-Kutta method (RK4)

In this project we are going to use the fourth order Runge-Kutta method. This method solves the integral in equation (18) with help of simpson's method. By applying this method we get

$$\int_{t_i}^{t_{i+1}} dt' \, v(t', x) = \frac{h}{6}[v(t_i, x_i) + 4v(t_{i+\frac{1}{2}}, x_{i+\frac{1}{2}}) + v(t_{i+1}, x_{i+1})] + \sigma(h^5) \tag{19}$$

By setting (19) in (18):

$$x_{i+1} = x_i + \frac{h}{6}[v(t_i, x_i) + 2v(t_{i+\frac{1}{2}}, x_{i+\frac{1}{2}}) + 2v(t_{i+\frac{1}{2}}, x_{i+\frac{1}{2}}) + v(t_{i+1}, x_{i+1})] + \sigma(h^5) \tag{20}$$

$x_{i+\frac{1}{2}}$ and x_{i+1} are unknown values. Now we define $k_1 = v(t_i, x_i)$, first approximate $x_{i+\frac{1}{2}}$ with Euler's method

$$x_{i+\frac{1}{2}} \approx x_i + \frac{h}{2} v(t_i, x_i) = x_i + \frac{h}{2} k_1 \tag{21}$$

and define

$$k_2 = v(t_{i+\frac{1}{2}}, x_i + \frac{h}{2} k_1) \tag{22}$$

Next we calculate $x_{i+\frac{1}{2}}$ with Eulers's method using k_2 instead of k_1 and then define

$$k_3 = v(t_{i+\frac{1}{2}}, x_i + \frac{h}{2} k_2) \tag{23}$$

We predict x_{i+1} with k_3 using Euler's method

$$x_{i+1} = x_i + h k_3 \tag{24}$$

$$k_4 = v(t_{i+1}, x_i + h k_3) \tag{25}$$

Now we can calculate our final x_{i+1} by inserting our k_j's in equation (20).

$$x_{i+1} = x_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) + \sigma(h^5) \tag{26}$$

2.5.2 RK4 in this Project

In section 2.5.1 it could be seen, that with the fourth order Runge-Kutta method, first-order ODE's (ordinary differential equations) can be solved. In this project however, we need to numerically solve the equation of motion for multiple particles, which is a second-order ODE (see equation (3)). This ODE can be split up into two first-order ODE's by writing

$$\frac{d\mathbf{r}}{dt} = \mathbf{v} \tag{27}$$

$$\frac{d\mathbf{v}}{dt} = G\sum_{j\neq i}^{n} m_j \frac{\mathbf{r}_{ij}}{r_{ij}^3} \tag{28}$$

Now we can use the RK4 method **r** and **v** simultaneously. Because we use this method in three dimensions and the fact, that we have a second-order ODE, the equations for the RK4 method look different than in section 2.5.1. The following steps give in the right order, how to approach the system, when we do a numerically calculation from the time t_i to t_{i+1}. It has to be noted, that when we do a calculation for one particle, directly after it, the same type of calculation gets performed for the other particles. The index on the upper left side of values refer to the number of the particle and the index on the upper right side gives the information, if the value is belogs to the RK4 method for the location or the velocity.

$$\mathbf{k}_1^r = \mathbf{v}_i \tag{29}$$

$$\mathbf{k}_1^v = \mathbf{a}({}^1\mathbf{r}_i, {}^2\mathbf{r}_i, ..., {}^N\mathbf{r}_i) \tag{30}$$

$$\mathbf{k}_2^r = \mathbf{v}_i + \frac{h}{2}\mathbf{k}_1^v \tag{31}$$

$$\mathbf{k}_2^v = \mathbf{a}({}^1\mathbf{r}_i + \frac{h}{2}\cdot{}^1\mathbf{k}_1^r, {}^2\mathbf{r}_i + \frac{h}{2}\cdot{}^2\mathbf{k}_1^r, ..., {}^N\mathbf{r}_i + \frac{h}{2}\cdot{}^N\mathbf{k}_1^r) \tag{32}$$

$$\mathbf{k}_3^r = \mathbf{v}_i + \frac{h}{2}\mathbf{k}_2^v \tag{33}$$

$$\mathbf{k}_3^v = \mathbf{a}({}^1\mathbf{r}_i + \frac{h}{2}\cdot{}^1\mathbf{k}_2^r, {}^2\mathbf{r}_i + \frac{h}{2}\cdot{}^2\mathbf{k}_2^r, ..., {}^N\mathbf{r}_i + \frac{h}{2}\cdot{}^N\mathbf{k}_2^r) \tag{34}$$

$$\mathbf{k}_4^r = \mathbf{v}_i + h\cdot\mathbf{k}_3^v \tag{35}$$

$$\mathbf{k}_4^v = \mathbf{a}({}^1\mathbf{r}_i + h\cdot{}^1\mathbf{k}_3^r, {}^2\mathbf{r}_i + h\cdot{}^2\mathbf{k}_3^r, ..., {}^N\mathbf{r}_i + h\cdot{}^N\mathbf{k}_3^r) \tag{36}$$

The accelerations can always get calculated by using equation (3). After the steps of equations (29) to (36) are executed, the velocity and positions of the particles for the time t_{i+1} can be calculated using formula (26).

3 Execution

In this project we are programming the motion of a n-body system. We are using two different algorithms for calculating the motion of the bodies called the Velocity-Verlet-algorithm and the Runge-Kutta-algorithm. We started to set up an algorithm for a two body system because this is much easier than dealing with n-bodies. However, we think it is sufficient to describe the program for n-bodies because the two body case is there also included. In our case, the bodies are represented by planets.

3.1 General structure and overview

We are using several classes in order to structure our program. The first class is called planet. Objects out of this class represent Planets with properties such as the position, velocity and mass. Originally, we thought that it would be nice to work with the planet objects and its positions and velocities in the algorithms itself. However, the Runge-Kutta method is heavily readable using this representation. Therefore, we used a matrix representation for position and velocity. The important information about position and velocity in kept in the matrix A. The second class is called solarsystem. This class contains all important functions and algorithms used to calculate the motion of the system. The header file for the basic functions looks as follows: (functions to calculate the Energy of the system (kinetic and potential) and to check the Virial theorem will be explained at a later point)

```
class solarsystem
{
public:
    //constructor
```

```
    solarsystem();
    //constants and arrays used for Energy-analysis
    double* vecKinpot;
    double* finalEnergy;
    vector<double> average_kin;
    vector<double> average_pot;
    //other constants and objects
    double R0=20;
    double averagemass=10;
    int numplanets;
    double G;
    double epsilon=0.002;
    vector<Planet> planets;
    Mat<double> A;
    Mat<double> dA;
    // use of the random class
    std::random_device rd;
    std::mt19937 gen;
    std::uniform_real_distribution<> dis;
    gaussian_random object;
    //general functions
    void addplanet(Planet Planet1);
    void addrandomplanet(double R_0);
    void setmatrices();
    //functions used only for Velocity-Verlet
    void VelocityVerlet(double dt,int n);
    void calculateForces();
    //Functions used only for Runge-Kutta-method
    void RungeKuttamethod(double dt,int n);
    mat derivate( Mat<double> B);
     // funktions for Energy-analysis
    void kinPotEnergy();   //VelocityVerlet RK-4
    void virial_output();
};
```

We will now briefly explain the use of the constructor, constants and functions of the class solarsystem. The functions, which are specific to Velocity-Verlet or Runge-Kutta method are explained in the section named after the method. The use of the random class itself will not be explained here. We are using a random number generator with a period of 2^{32}. This is sufficient. The random number generator is seeded by default while constructing a object of solarsystem.

3.1.1 constructor

We need a constructor in order to create objects of the class. With constructing the class, we also initialise the random number generator and set the number of planets of the created solarsystem-object to 0.

```
solarsystem::solarsystem() : gen(this->rd()), dis(0,1)
{
    this->numplanets=0;
}
```

3.1.2 constants and objects

We need some constants for the random number generator in order to create random planet-objects with the right properties. R0 is the radius for the sphere in which the planets should be distributed with constant particle density. The average-mass (averagemass) is the averagemass of the random planet-objects. The standard deviation of the planet objects is manually set to a specific value in the function addrandomplanet(double R0). The integer numplanets counts the actual number of planets of a specific planet object. The gravitational constant G is in our case dependent on the number of planets and the average-mass of a planet. Therefore, this constant is recalculated every time you add a planet. (in the function addrandomplanet) The smoothing-factor epsilon is also initialised and defined.

3.1.3 general functions

Our n-body- system can be used in different ways. You can manually add a planet with chosen initial position, initial velocity and mass. This is done by the function addplanet(Planet Planet1). Therefore, this algorithm can also be used to calculate the motion of a given two body system. With small changes, it is also possible to simulate our solar-system. (G has to be changed)

```
void solarsystem::addplanet(Planet Planet1){
    planets.push_back(Planet1); // adds the planet to the vector of planets
    average_kin.push_back(0);
    average_pot.push_back(0);
    this->numplanets+=1;
    G=4*M_PI*M_PI*R0*R0*R0/(32*this->numplanets*averagemass);
}
```

On the other way, you can choose to add a planet with random position within a sphere and with given Gaussian-distributed mass of the planet. This is done by the function addrandomplanet(double R0). We set the planet-velocities to 0. With Box–Muller transform, we get a normal-distribution with given mean-value and given standard-deviation. The mass is set to a random Gaussian-distributed value. We wand a equal density at t=0 within a sphere. Therefore, we have to manipulate a random generated triple (u,v,w between 0 and 1) as follows:

$$r = R0 \cdot \sqrt[3]{u}$$
$$\theta = \arccos(1-2v)$$
$$\phi = 2\pi\omega$$

We want to use spherical coordinate. Therefore, we transform back to spherical coordinates, simplify and get:

$$x = R0 \cdot \sqrt[3]{u} \cdot \sqrt{1-(1-2\cdot v)^2} \cdot cos(2\pi\cdot\omega)$$
$$y = R0 \cdot \sqrt[3]{u} \cdot \sqrt{1-(1-2\cdot v)^2} \cdot sin(2\pi\cdot\omega)$$
$$z = R0 \cdot \sqrt[3]{u} \cdot (1-2\cdot\omega)$$

In the end, we add the new planet with the function addplanet to the solarsystem.

```
void solarsystem::addrandomplanet(){
    Planet randomplanet;
    // set andom mass with Gaussian-distribution
    randomplanet.m=object.generateGaussianNoise(10,1);
    // set velocities to 0
    randomplanet.velocity[0]=0;
    randomplanet.velocity[1]=0;
    randomplanet.velocity[2]=0;
    //generate random position
    double r=dis(gen);
    double theta=dis(gen);
    double phi=dis(gen);
    //transform random position in order to get
    //Cartesian coordinates with constant density
    randomplanet.position[0]=R0*pow(r,(1.0/3.0))*sqrt((1-(pow(1-2*theta,2))))*cos(2*M_PI*phi);
    randomplanet.position[1]=R0*pow(r,(1.0/3.0))*sqrt((1-(pow(1-2*theta,2))))*sin(2*M_PI*phi);
    randomplanet.position[2]=R0*pow(r,(1.0/3.0))*(1-(2*theta));
    addplanet(randomplanet);
}
```

To use the matrix-representation of position and velocity, we wrote the function setmatrices(), which constructs a martix A, which contains all positions and velocities of the planets, which has been added to the system so far. The position of the planets can be found in the first column and the velocity in the second column. Every planet has 3 Space-Coordinates and 3 Velocity-Coordinates. In addition to that, setmatrices() constructs the matrix dA, which is the derivative of the matrix A. This means that in the first column are velocities and in the second column are accelerations. For RK4, this is really useful because we can perform very understandable operations like $A = A+dA\cdot dt$.

Notice, that the velocity sits in the second column of A as well as in the first column of dA.

$$A = \begin{bmatrix} r_{x_1} & v_{x_1} \\ r_{y_1} & v_{y_1} \\ r_{z_1} & v_{z_1} \\ r_{x_2} & v_{x_2} \\ \dots & \dots \\ r_{x_n} & v_{x_n} \\ r_{y_n} & v_{y_n} \\ r_{z_n} & v_{z_n} \end{bmatrix} \qquad dA = \begin{bmatrix} v_{x_1} & a_{x_1} \\ v_{y_1} & a_{y_1} \\ v_{z_1} & a_{z_1} \\ v_{x_2} & a_{x_2} \\ \dots & \dots \\ v_{x_n} & a_{x_n} \\ v_{y_n} & a_{y_n} \\ v_{z_n} & a_{z_n} \end{bmatrix} \tag{37}$$

```
void solarsystem::setmatrices(){
    A=Mat<double>(this->numplanets*3,2);
    dA=Mat<double>(this->numplanets*3,2);

    for(int i=0;i<this->numplanets;i+=1){
        for(int j=0;j<3;j++){
            A(3*i+j,0)=planets[i].position[j];
            A(3*i+j,1)=planets[i].velocity[j];
        }
    }

    for(int i=0;i<this->numplanets;i+=1){
        for(int j=0;j<3;j++){
            dA(3*i+j,0)=planets[i].velocity[j];
            dA(3*i+j,1)=planets[i].force[j]/planets[i].m;
        }
    }
}
```

3.2 Velocity-verlet-algorithm

In the following, we find the central steps for implementing the Velocity-verlet-algorithm. This is a non-selfstarting algorithm because we need to calculate v(t+0.5dt) before entering the loop of the algorithm. We are using the velocity in the first column of the matrix A for calculations.

```
void solarsystem::VelocityVerlet(double dt,int n){
  ...
   setmatrices();
   calculateForces();
   A.col(1)=A.col(1)+0.5*dt*dA.col(1);  // v(t)->v(t+0.5dt)
   for(int k=0;k<n;k++){
      A.col(0)=A.col(0)+A.col(1)*dt;    // r(t+dt)
       calculateForces();               // a(t+dt)
      A.col(1)=A.col(1)+dt*dA.col(1);   // v(t+0.5dt) -> v(t+3/2dt)
       for(int i=0;i<this->numplanets;i+=1){
      dA.col(0)=A.col(1);  //updating velocities in dA, not necessary
        }
       }
  ...
}
```

The function calculateForces() calculates the forces of all the planets and with that the acceleration and writes it into the matrix dA. The forces of the planetobjects are primarily used for local calculations.

```
void solarsystem::calculateForces(){


    for(int k=0;k<this->numplanets;k++){
        for(int l=0;l<3;l++){
            planets[k].force[l]=0;
        }
```

```
    }
    for(int i=0;i<this->numplanets;i++){
        for(int j=i+1;j<this->numplanets;j++){
            double dx=A(3*i,0)-A(3*j,0);
            double dy=A(3*i+1,0)-A(3*j+1,0);
            double dz=A(3*i+2,0)-A(3*j+2,0);
            double dr2=dx*dx+dy*dy+dz*dz+epsilon*epsilon;
            //calculate forces
            double Fx=(G*(planets[i].m)*(planets[j].m)*dx)/pow(dr2,1.5);
            double Fy=(G*(planets[i].m)*(planets[j].m)*dy)/pow(dr2,1.5);
            double Fz=(G*(planets[i].m)*(planets[j].m)*dz)/pow(dr2,1.5);
            //update planet properties
            planets[i].force[0]-=Fx;
            planets[j].force[0]+=Fx;
            planets[i].force[1]-=Fy;
            planets[j].force[1]+=Fy;
            planets[i].force[2]-=Fz;
            planets[j].force[2]+=Fz;
        }
    }
    //update the matrix dA
    for(int i=0;i<this->numplanets;i+=1){
        for(int j=0;j<3;j++){
            dA(3*i+j,1)=planets[i].force[j]/planets[i].m;
        }
    }

}
```

3.3 Runge Kutta-Method

We use the matrix representation for the RK4-Method. We first calculate the derivatives of the matrix A at different times and store them in the matrices k1 to k4. Afterwards, we simply numerically integrate using the coefficients given by the formula of RK4. In this representation, the algorithm is very compact and easily readable.

```
void solarsystem::RungeKuttamethod(double dt,int n){
    setmatrices();
    Mat<double> k1,k2,k3,k4;
    k1=Mat<double>(this->numplanets*3,2);
    k2=Mat<double>(this->numplanets*3,2);
    k3=Mat<double>(this->numplanets*3,2);
    k4=Mat<double>(this->numplanets*3,2);

    for(int i=0;i<n;i++){
        k1=derivate(A);
        k2=derivate(A+k1*(dt/2));
        k3=derivate(A+k2*(dt/2));
        k4=derivate(A+k3*dt);
        A=A+(1.0/6.0)*(k1+2*k2+2*k3+k4)*dt;
        }
    }
}
```

The function derivate(Mat<double> B) calculates the derivative of the matrix in the argument B. The function uses the positions stored in the matrix B to calculate the forces of the planets. Finally, we can calculate the accelerations on the planets using their mass. We write the velocity of the first column of B into the 0th column of dB and we write the calculated accelerations in the first column of dB. In total, we calculated and return the derivative of the input matrix B.

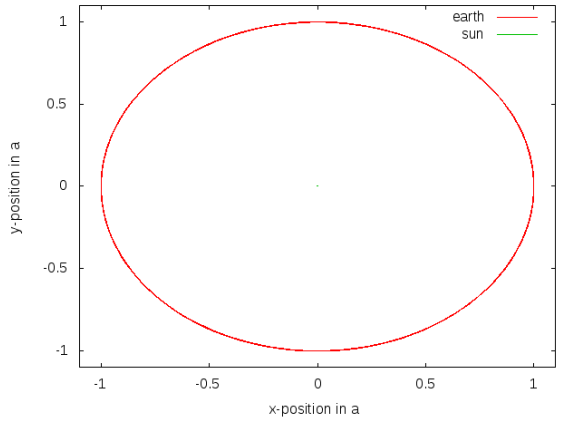
```
mat solarsystem::derivate( Mat<double> B){
    Mat<double> dC;
```

```
    dC=Mat<double>(this->numplanets*3,2,fill::zeros);
    for(int i=0;i<this->numplanets;i++){
        for(int j=i+1;j<this->numplanets;j++){
            double dx=B(3*i,0)-B(3*j,0);
            double dy=B(3*i+1,0)-B(3*j+1,0);
            double dz=B(3*i+2,0)-B(3*j+2,0);
            double dr2=dx*dx+dy*dy+dz*dz+epsilon*epsilon;
            //calculate forces
            double Fx=(G*(planets[i].m)*(planets[j].m)*dx)/pow(dr2,1.5);
            double Fy=(G*(planets[i].m)*(planets[j].m)*dy)/pow(dr2,1.5);
            double Fz=(G*(planets[i].m)*(planets[j].m)*dz)/pow(dr2,1.5);
            dC(3*i,1)-=Fx;
            dC(3*j,1)+=Fx;
            dC(3*i+1,1)-=Fy;
            dC(3*j+1,1)+=Fy;
            dC(3*i+2,1)-=Fz;
            dC(3*j+2,1)+=Fz;
        }
    }
    for(int i=0;i<this->numplanets;i+=1){
        for(int j=0;j<3;j++){
            dC(3*i+j,1)=dC(3*i+j,1)/planets[i].m;
        }
    }
    dC.col(0)=B.col(1);
    return(dC);//return dC with velocity from B and new acceleration
}
```

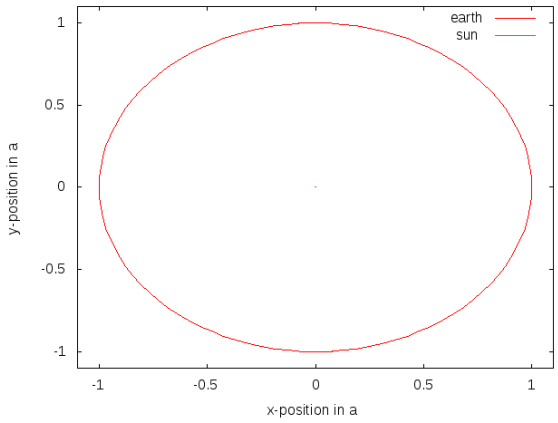
4 Results

4.1 Results of the two-body-system

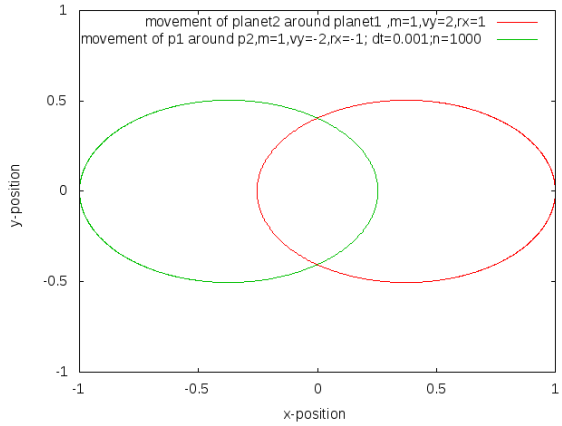
We first test our two body algorithm with well known systems in order to check whether we can reproduce our predictions. We simulate the earth and the sun. With Velocityverlet (Verlet), we get figure 1a and with Rung Kutta method (RK4), we get figure 1b. Both trajectories fit to what we expect from the system. We see a circular motion. We can't see a qualitative difference of the two calculated trajectories. Next, we test our two-body solver with a bit more complicated initial conditions. The initial conditions can be looked up in the figures itself. We get for Verlet the figure 1c and for RK4 the figure 1d. We get the qualitative correct movement of both planets and we can't see qualitative differences of the two solvers.



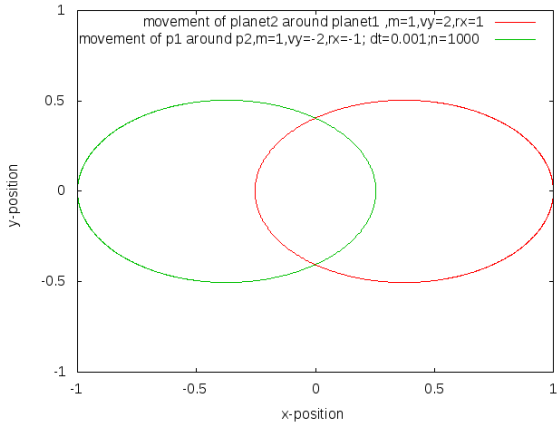
(a) sun-earth system simulated for 10y with Verlet-method



(b) sun-earth-system simulated for 10y with RK4-method



(c) two-body-problem solved with Verlet

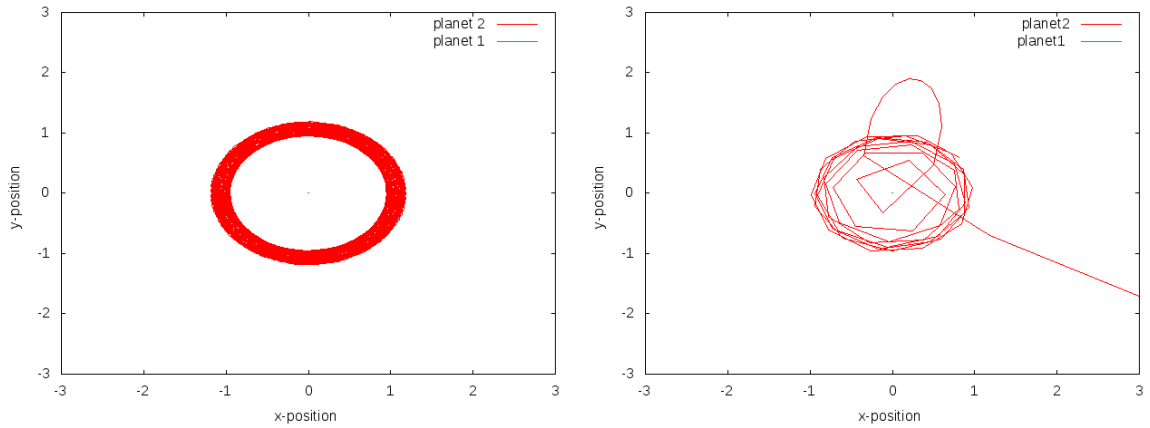


(d) two-body-problem solved with RK4

Figure 1: two-body-problem-plots

Since we use two different solvers, we want to know how they differ with respect to large time-steps and long times in order to say which of them should be used for which case. For fairly large time-steps, we clearly saw the difference. With time-steps of 35.6 days the Verlet-algorithm in figure 2a is more stable than the RK4-algorithm in figure 2b. Using RK4, the earth is in the end not bound any more. For large time-steps it seems therefore to be better to use Verlet-algorithm.

However, it was hard to find out a difference of the two solvers for very long times. It seemed to be as if they were both accurate. We decided to look at the Energy-conservation of the two systems. We simulates 10000 years (dt=0.001 n=10000000) with both methods. The Verlet-algorithm was much faster than the RK4-algorithm. This will be important when we decide which we use for the n-body-problem. The plot of the trajectories of both methods look qualitatively good. That means that both algorithms are stable. In both methods, the earth orbits the sun in a stable orbit. In order to say something about the accuracy of the algorithms, we looked at the total energy of the system. We calculated the quotient of the total energy of the system at the end of the calculations and the total energy of the system at the beginning of the calculations. (a corresponds to 1 year and au to 1 astronomical unit) We found out that for large times, RK4-algoritm is a bit more precise than the Verlet-algorithm. However, RK4 is numerically much more expensive.(comparing FLOPS and computation time) (see table 1 for Energy analysis)



(a) sun-earth system simulated with Verlet-method; positions in earth-sun-distances

(b) sun-earth-system simulated with RK4-method; positions in earth-sun-distances

Figure 2: analysis of large time-steps for the two-body-problem of the sun (planet1) and the earth (planet2) for dt=0.1 and n=1000; This is equivalent to 100 years with time-steps of approximately 36.5 days

Table 1: Analysis of energy conservation of Verlet/RK4-algorithm for dt=0.01 and n=10000000. This corresponds to 10000 years. We are comparing the total energy in the beginning wit the total energy in the end. Before doing that, we convinced ourselves that both methods are stable

	Verlet	RK4
$E_{tot,i}$ in $\frac{M_o au^2}{a^2}$	5.92174 10^{-5}	5.92174 10^{-5}
$E_{tot,f}$ in $\frac{M_o au^2}{a^2}$	5.91593 10^{-5}	5.92173 10^{-5}
$\frac{E_{tot,f}}{E_{tot,i}}$	0.99902	0.99999

4.2 Conclusion of the results of the two-body-system

We recommend for large time-steps the Verlet-algorithm because the trajectory calculated with Verlet was more stable than the trajectory calculated with RK4-method. For long times, we found out that the RK4-algorithm is a bit more accurate than the Verlet-algorithm. We derived this result from looking at the energy conservation. However, the RK4-algorithm is numerically more expensive than the Verlet-algorithm. Therefore, it could also make sense to run programs with higher numbers of n with the Verlet-algorithm.

5 Discussion

6 Source code

References

[1] Source: `https://en.wikipedia.org/wiki/Open_cluster`, date: 10.12.2015 time: 17:00

[2]