UNIVERSITY OF OSLO

COMPUTATIONAL PHYSICS

Project 5

Authors:

Birgitte Madsen, 66

Magnus Isaksen, 14

Soumya Chalakkal, 51

Autumn 2015

UiO : University of Oslo

Department of Physics
University of Oslo
Sem Sælands vei 24
0371 Oslo, Norway
+47 22 85 64 28
http://www.mn.uio.no/fysikk/english/

Course:

Computational Physics

Project number:

5

Link to GitHub folder:

`https://?????`

Hand-in deadline:

Friday, December 11, 2015

Project Members:

Birgitte Madsen, 66
Magnus Isaksen, 14
Soumya Chalakkal, 51

Copies: 1
Page count: 21
Appendices: 0
Completed: ????, 2015

Abstract

TABLE OF CONTENTS

CHAPTER

1

INTRODUCTION

CHAPTER

2 METHOD

The source codes for the algorithms described in this chapter can be found in the Github folder `https://?????`. [1]

2.1 Transformation between units

When considering at a planetary system or a larger system like a galaxy it is inconvenient to use the SI units for length and time. Instead, to investigate the evolution of astronomical systems, it is an advantage to use days, years (yr) or even longer time periods as the unit of time, and astronomical units (AU) or light years (ly) as the unit of distance. The change in unit system, evidently changes the considered constant, the gravitational constant G, which in SI units is given as

$$G = 6.67 \cdot 10^{-11} \frac{\mathrm{Nm}^2}{\mathrm{kg}^2} \tag{2.1}$$

For a planetary system like the Earth-Sun system it is better to consider distances in AU instead of meters, and use days as a measure of time, as the planet doesn't move far on its orbit in a second. Furthermore, it is an advantage to express the masses in the system in units of solar masses. Hence, the constants have to be transformed into these unit systems.

The gravitational constant G is transformed using

$$1\ \mathrm{AU} = 1.495 \cdot 10^{11}\ \mathrm{m} \qquad \text{and} \qquad 1\ \mathrm{M}_\odot = 1.989 \cdot 10^{30}\ \mathrm{kg} \tag{2.2}$$

giving the gravitational constant

$$G = 2.96 \cdot 10^{-4} \frac{\mathrm{AU}^2}{\mathrm{days}^2 \mathrm{M}_\odot} \tag{2.3}$$

which is convenient when considering a planetary system.

For a star cluster, the distances are greater and the time scales are larger than in the planetary system. Hence, it is more convenient to use years as the unit of time and lightyears as the unit of distance.

$$1\ \mathrm{yr} = 3.1536 \cdot 10^7 \mathrm{s} \qquad \text{and} \qquad c = 2.008 \cdot 10^8 \frac{\mathrm{m}}{\mathrm{s}} \tag{2.4}$$

[1] FiXme Note: fix these lines

in which c is the speed of light. This yields that 1 ly is

$$1 \text{ ly} = 9.45 \cdot 10^15 \text{ m} \tag{2.5}$$

Giving the gravitational constant

$$G = 1.536 \cdot 10^{-13} \frac{\text{ly}^2}{\text{yr}^2 \text{M}_\odot} \tag{2.6}$$

2.2 Newtonian two-body problem in three dimension

The problem of solving the time-evolution of a two-body system in three dimensions can reasonably be considered in two different coordinate systems: one coordinate system with one of the bodies in rest compared to the frame of reference in which the other body is moving, and one coordinate system with both of the bodies moving relative to the frame of reference. Both of these reference systems are depicted in Fig. 2.1.

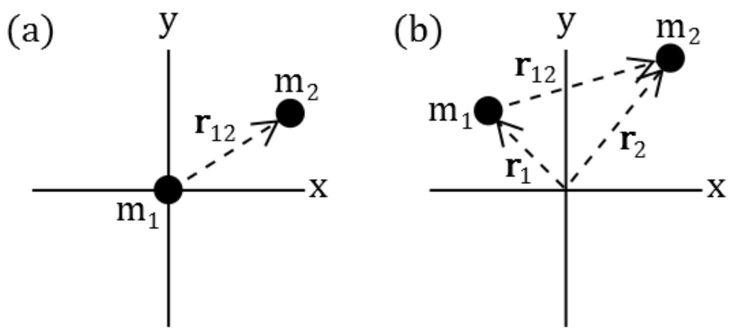


Figure 2.1. Two-dimensional illustration of the three-dimensional problem of determining determining the relative distance and relative velocity between two bodies. In (a) body 1 with mass m_1 is considered stationary in position- and velocity-space, whilst body 2 with mass m_2 moves relative to body 1. In (b) both body 1 and 2 moves relative to the frame of reference in position and time, yielding that the position vector between body 1 and 2 is given as $\mathbf{r}_{12} = \mathbf{r}_2 - \mathbf{r}_1$.

In the codes presented in this section, solving the problem in coordinate system (a) will first be considered for simplicity. Thereafter, the codes will be extended to include the movement of body 1 relative to the coordinate system, since this will be useful when extending the codes to an N body system.

In the problem, $\mathbf{r}(t)$ is the three-dimensional space vector consisting of the coordinated $(x(t), y(t), z(t))$, whilst $\mathbf{v}(t)$ is the three-dimensional velocity vector with coordinates $(v_x(t), v_y(t), v_z(t))$, both of which are dependent on time.

In general, the considered differential equation is

$$\frac{dy}{dt} = f(t, y) \tag{2.7}$$

Which yields that

$$y(t) = \int f(t, y) dt \tag{2.8}$$

[2] For the two bodies in a three dimensional Newtonian gravitational field this corresponds to six coupled differential equations given by the vector equations

$$\frac{d\mathbf{r}}{dt} = \mathbf{v} \qquad \text{and} \qquad \frac{d\mathbf{v}}{dt} = -\frac{GM_1M_2}{r^3}\mathbf{r} \tag{2.9}$$

[3] in which M_1 and M_2 [4] are the masses of the two bodies, respectively, whilst r is the distance between the bodies. The equations in (2.9) are computed by the script given below in which *drdt* corresponds to the derivative of the coordinates of the position, and *dvdt* corresponds to the derivative of the velocity coordinates.

```
void Derivative(double r[3], double v[3], double (&drdt)[3], double (&dvdt)[3], double
    G, double mass){
    drdt[0] = v[0];
    drdt[1] = v[1];
    drdt[2] = v[2];
    double distance_squared = r[0]*r[0] + r[1]*r[1] + r[2]*r[2];
    double newtonian_force = -G*mass/pow(distance_squared,1.5);
    dvdt[0] = newtonian_force*r[0];
    dvdt[1] = newtonian_force*r[1];
    dvdt[2] = newtonian_force*r[2];
}
```

When including movement of both bodies relative to the frame of reference, the *Derivative* function must be slightly modified, since then the relative position of the two bodies will be given as $\mathbf{r}_{12} = \mathbf{r}_2 - \mathbf{r}_1$. For a general case with N particles, the *distance_ squared* between body i and j, and the acceleration in x, y and z due to the Newtonian force between body i and j can be determined by the following lines of code.

```
for (int j=0; j<number_of_particles; j++)
    {
        if (j!=i)
        {
            distance_squared = 0;
            for (int k=0; k<3; k++)
            {
                distance_squared += (r(i,k)-r(j,k))*(r(i,k)-r(j,k));
            }
            force_between_particles = m(j) * pow(distance_squared,-1.5);
            acc_x += G*force_between_particles*(r(j,0)-r(i,0));
            acc_y += G*force_between_particles*(r(j,1)-r(i,1));
            acc_z += G*force_between_particles*(r(j,2)-r(i,2));
        }
    }
```

The if statement in the for loop over all bodies adds up the acceleration in all three dimensions of all particles due to the presence of other particles. Hence, for the two body problem, the argument in the if statement will only be true once for each of the two particles.

[2] FiXme Note: do we need to write y_{i+1} eq from p. 250 in lecture notes??
[3] FiXme Note: maybe we should divide by mass as on p. 248??
[4] FiXme Note: fix the this with M_1 and M_2

2.2.1 Velocity-Verlet method

- Remember to write about accuracy of algorithm!!

Consider the Taylor expansion of the vector function $\mathbf{r}(t_i \pm \delta t)$:

$$\mathbf{r}(t_i \pm \delta t) = \mathbf{r}(t_i) \pm \mathbf{v}(t_i)\delta t + \mathbf{a}(t_i)\frac{\delta t^2}{2} \pm \frac{\delta t^3}{6}\frac{d^3\mathbf{r}(t_i)}{dt^3} + \mathscr{O}(\delta t^4) \tag{2.10}$$

Adding the two expressions in Eq. (2.10) gives

$$\mathbf{r}(t_i + \delta t) = 2\mathbf{r}(t_i) - \mathbf{r}(t_i - \delta t) + \mathbf{a}(t_i)\delta t^2 + \mathscr{O}(\delta t^4) \tag{2.11}$$

which has a truncation error that goes as $\mathscr{O}(\delta t^4)$.

$$\mathbf{r}(t_i + \delta t) = \mathbf{r}(t_i) + \mathbf{v}(t_i)\delta t + \frac{1}{2}\mathbf{a}(t_i)\delta t^2 \tag{2.12}$$

$$\mathbf{v}(t + \delta t) = \mathbf{v}(t) + \frac{1}{2}(\mathbf{a}(t) + \mathbf{a}(t + \delta t))\delta t \tag{2.13}$$

The velocity is in the algorithm calculated [5] by first calculating

$$\mathbf{v}_{part1}(t + \delta t) = \mathbf{v}(t) + \frac{1}{2}\mathbf{a}(t)\delta t \tag{2.14}$$

and then use ?? [6] to determine $\mathbf{a}(t + \delta t)$, which is then used to compute the remaining term of Eq. (2.13) as

$$\mathbf{v}_{part2}(t + \delta t) = \frac{1}{2}\mathbf{a}(t + \delta t)\delta t \tag{2.15}$$

The velocity-Verlet method uses the algortihm *Derivative* described in Sec. 2.2, to generate the six differential equations, in the following while-loop that runs until reaching the final time in time steps of length $\delta t = (t_{initial} - t_{final})/(\#timesteps)$.

```
while(time<=t_final){
Derivative(r,v,drdt,dvdt,G,mass);
for(int i=0; i<6 ; i++){
r[i] = r[i]+dt*drdt[i] + 0.5 * dt * dt * dvdt[i];
v_partly[i] = drdt[i] + 0.5 * dt * dvdt[i];
dvdt[i] = v_partly[i];
}
Derivative(r,v,drdt,dvdt,G,mass);
for(int i=0; i<n ; i++){
v[i] = v_partly[i] + 0.5 * dt * dvdt[i];
}
time += dt;
}
```

[5] FiXme Note: ad to gange
[6] FiXme Note: fix this!

2.2.2 Fourth Order Runge-Kutta Method

- Remember to write about accuracy of algorithm!!

The Runge-Kutta method is based on Taylor expansions, with the next function value after a times step $\delta t = t_i - t_{i+1}$ being computed from four more or less improved slopes of the function in the points t_i, $t_i + \delta t/2$ and t_{i+1}.

The first step of the RK4 method is to compute the slope k_1 of the function in t_i by

$$k_1 = \delta t f(t_i, y_i)$$

Then the slope k_1 at the midpoint is computed from k_1 as

$$k_2 = \delta t f(t_i + \delta t/2, y_i + k_1/2)$$

The slope at the midpoint is then improved from k_2 by

$$k_3 = \delta t f(t_i + \delta t/2, y_i + k_2/2)$$

from which the slope k_4 at the next step y_{i+1} is predicted to be

$$k_4 = \delta t f(t_i + \delta t, y_i + k_3)$$

From the computed slopes k_1, k_2, k_3 and k_4, the function value at $t_i + \delta t$ is computed as

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{2.16}$$

When implementing this for the two-body problem in three dimensions, it boils down to a continuous call of two functions, namely the function *Derivative* given in Sec. 2.2 and the function *updating_dummies* given below.

```
void updating_dummies(double dt, double drdt[3], double dvdt[3], double (&r_dummy)[3],
    double (&v_dummy)[3], double number, double (&kr)[3], double (&kv)[3], double
    r[3], double v[3])
{
    for (int i = 0; i<3; i++){
        kr[i] = dt * drdt[i];
        kv[i] = dt * dvdt[i];
        r_dummy[i] = r[i] + kr[i]/number;
        v_dummy[i] = v[i] + kv[i]/number;
    }
}
```

The function *updating_dummies* computes the values of k_1, k_2, k_3 and k_4 for all three space coordinates and velocity coordinates from the derivatives *drdt* and *dvdt* computed by the *Derivative* function. To compute the next step given by Eq. (2.16), the following succession of function calls are made until the time reaches the final time t_{final} after $(t_{final} - t_{inital})/\delta t$ time steps.

```
while(time<=t_final){
    Derivative(r,v,drdt,dvdt,G,mass);
updating_dummies(dt,drdt,dvdt,r_dummy,v_dummy,2,k1r,k1v,r,v);
Derivative(r_dummy,v_dummy,drdt,dvdt,G,mass);
updating_dummies(dt,drdt,dvdt,r_dummy,v_dummy,2,k2r,k2v,r,v);
Derivative(r_dummy,v_dummy,drdt,dvdt,G,mass);
updating_dummies(dt,drdt,dvdt,r_dummy,v_dummy,1,k3r,k3v,r,v);
Derivative(r_dummy,v_dummy,drdt,dvdt,G,mass);
for (int i = 0; i<n; i++){
    k4r[i] = dt*drdt[i];
    k4v[i] = dt*dvdt[i];
}
for (int i=0; i<n;i++){
    r[i] = r[i] +(1.0/6.0)*(k1r[i]+2*k2r[i]+2*k3r[i]+k4r[i]);
    v[i] = v[i] +(1.0/6.0)*(k1v[i]+2*k2v[i]+2*k3v[i]+k4v[i]);
}
time += dt;
}
```

When including the movement of both bodies relative to the reference system or adding more bodies to the system, **r**'s, **v**'s, **k**'s etc. must be generated for all of the particles, yielding introduction of a for loop over all particles.

2.3 Generating Position, Mass and Velocity for Cluster Particles

[7] [8] Write here, what kind of distribution we want for the variables!!!

2.3.1 Gaussian Distributed Mass

[9]

Pseudo-random numbers, corresponding to masses, randomly distributed by a Gaussian distribution is generated following the Box-Muller transform. The basic form of the Box-Muller transform gives

$$X_1 = \sqrt{-2ln(V_1)}sin(2V_2) = Rcos(\theta)$$
$$X_2 = \sqrt{-2ln(V_1)}cos(2V_2) = Rsin(\theta)$$

where $R^2 = -2ln(V_1)$ and $\theta = 2V_1$. X_1 and X_2 are random numbers distributed according to a Gaussian distribution of mean 0 and variance 1. In polar form the above two equations becomes

$$X_1 = u\sqrt{\frac{-2ln(s)}{s}}$$
$$X_2 = v\sqrt{\frac{-2ln(s)}{s}}$$

where $s = R^2 = u^2 + v^2$. [10] Here u and v are uniformly distributed in the interval [-1,1] and points only within the unit circle is admitted (see Fig. 2.2). Therefore, only those pairs of u and v which gives a value

[7]FiXme Note: write small intro
[8]FiXme Note: in this section, we can introduce a generation of velocity, if we need that at some point
[9]FiXme Note: we should probably have a ref. here
[10]FiXme Note: how the hell??

for s in the interval (0,1) is considered. Value of s is similar to that of V_1 and $\theta/2\pi$ is similar to that of V_2 in the basic form.

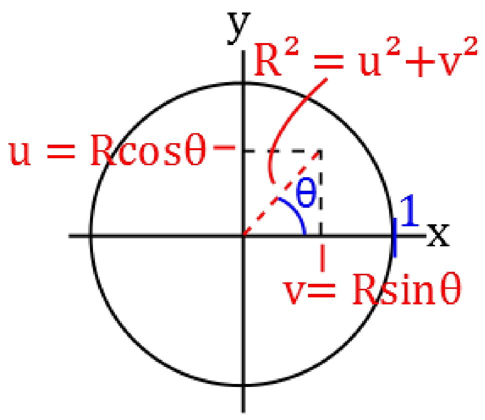


Figure 2.2. ???

The lines of code below shows the implementation of the generation of masses normally distributed with a mean of 10M$_\odot$ and a standard deviation of 1M$_\odot$.

```
void gaussian_mass_generator(vec (&mass), int number_of_particles)
{
  srand(time(NULL));
  for (int i = 0; i < number_of_particles; i++)
  {
  static int iset = 0;
  static double gset;
  double fac, rsq, v1, v2;
    do{
   //generate two random numbers uniformly distributed in the interval [-1,1]
      v1 = 2.*((double) rand() / (RAND_MAX)) -1.0;
      v2 = 2.*((double) rand() / (RAND_MAX)) -1.0;
      //Radius of the numbers (within the unit circle) squared
      rsq = v1*v1+v2*v2;
    } while (rsq >= 1.0 || rsq == 0.);
    //computing the gaussian distributed numbers
    fac = sqrt(-2.*log(rsq)/rsq);
    gset = v1*fac;
    iset = 1;
    mass(i) = v2*fac;
    mass(i) += 10;
  }
}
```

In the function *gaussian_ mass_ generator*, v1 and v2 are random numbers uniformly distributed in the interval [-1,1]. $rsq = v1^2 + v2^2$ corresponds to $s = R^2 = u^2 + v^2$. Using the do-while loop only those pairs of v1 and v2 that produces an s equal to 0 or greater than or equal to 1 is generated so that the points are inside unit circle. Variable gset correspond to X1.

To test whether the generated masses are actually normally distributed around $10M_\odot$ with a standard deviation of $1M_\odot$, 100,000 masses are generated, by the presented code, and plotted in a histogram below.

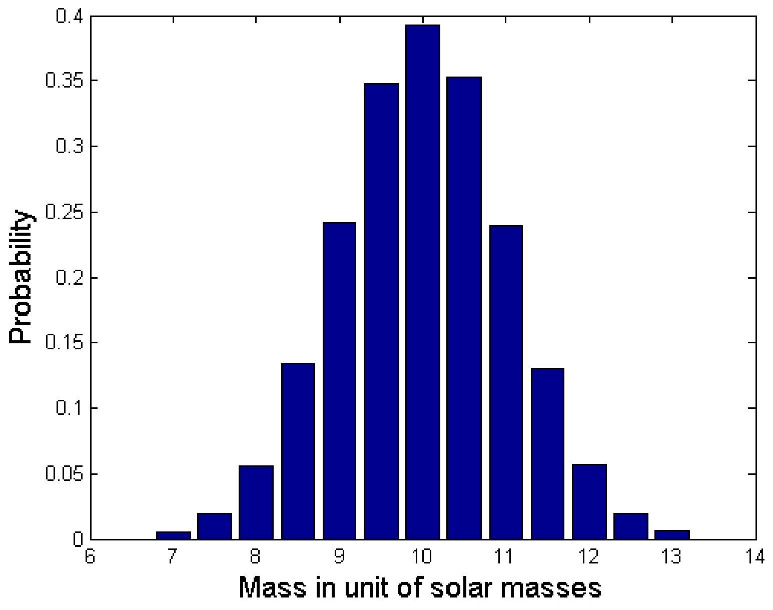


Figure 2.3. Histogram of the mass of 100,000 particles generated by the c++ code introduced [11].

[12]

2.3.2 Uniformly Distributed Position

[13] To generate positions uniformly distributed inside a sphere of radius R_0, random numbers generated using the *rand* function in c++ are used and converted into coordinates of uniformly distributed particles within the sphere. In order to get a uniform density of particles three variables v, w and u corresponding to random numbers uniformly distributed between 0 and 1 are introduced. The spherical coordinates θ, ϕ and r are then linked to these variables using the equation

$$\theta = cos^{-1}(1-2v)$$
$$\phi = 2\pi w$$
$$r = R_0(u)^{1/3}$$

The following equations are then used to get back to the Cartesian coordinate system.

$$x = rsin(\theta)cos(\phi)$$
$$y = rsin(\theta)sin(\phi)$$
$$z = rcos(\theta)$$

After performing these steps, a uniform distribution of N particles within a sphere of radius R_0 is achieved. Below, the code for generating this uniform distribution within a sphere of radius $R_0 = 20$ly is introduced.

[12]FiXme Note: eq. include gaussian dist. in fig
[13]FiXme Note: ref to the notes!!

```
void uniform_pos_generator(mat (&position), int N)
{
double pi=3.14159, c = 2*pi, R = 20;
vec phi(N), r(N), theta(N), x(N), y(N), v(N);

srand(time(NULL));

for (int i=0;i<N;i++){

       x(i) = ((double) rand() / (RAND_MAX)); //random numbers generated in the
           interval(0,1)
       y(i) = ((double) rand() / (RAND_MAX));
       v(i) = ((double) rand() / (RAND_MAX));
   }
for (int i=0;i<N;i++){
       phi(i)=c*x(i);
       r(i)=R*pow(y(i),1.0/3.0);
       theta(i)=acos(1.0-2.0*v(i));
       position(i,0)=r(i)*sin(theta(i))*cos(phi(i));
       position(i,1)=r(i)*sin(theta(i))*sin(phi(i));
       position(i,2)= r(i)*cos(theta(i));
   }
}
```

To test whether the generated positions within the sphere of radius 20 ly, the density of particles in the cross-sectional area of each x-value is determined and plotted as a histogram in Fig. 2.5 for $100,000$ particles with position generated by the introduced lines of code. The density of particles in the cross-sectional area of each x-value is found by dividing the total number of particles with that x-value with the cross-sectional area of the sphere in that x-value (see Fig. 2.4).

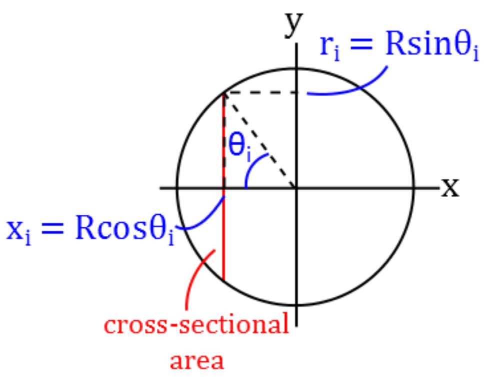


Figure 2.4. Two-dimensional illustration of the three-dimensional problem of determining the density of particles in each x-value.

The cross-sectional area of the sphere in a specific area is found from a little trigonometry, by first considering that the radius of the circle that makes of the cross-sectional area in a point x_i is given by $r_i = 20sin\theta_i$ ly. This yields that the area A_i of the cross-sectional area, in ly, is given as

$$A_i = 400\pi sin^2\theta_i = 400\pi(1-cos^2\theta) \tag{2.17}$$

in which the last equal sign stems from $1 = cos^2\theta + sin^2\theta$. But $x_i = 20cos\theta_i$ ly, giving

$$A_i = \pi(400 - x_i^2) \tag{2.18}$$

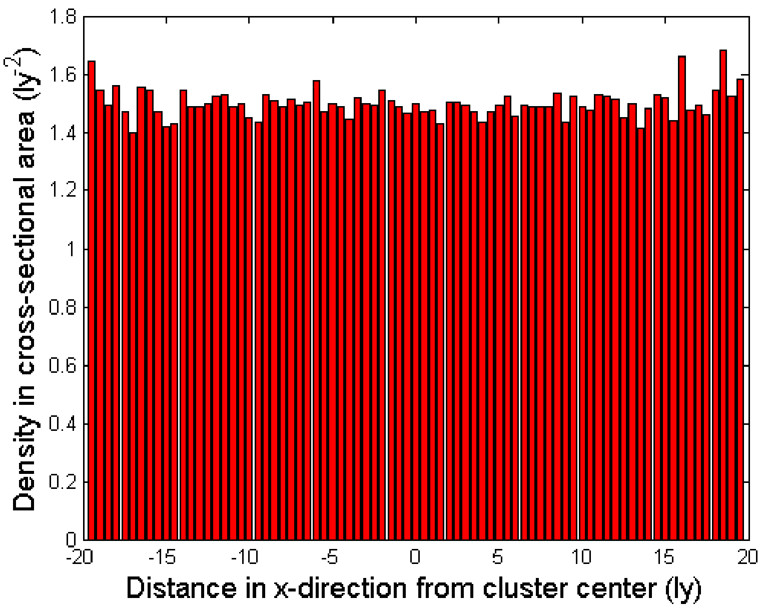


Figure 2.5. Histogram of density of 100,000 particles with position generated by the code introduced in [14] as a function of the x-coordinate of the particles. The histogram is made with bins in the interval $[-19.5; 19.5]$ and a bin-size of 0.5. The distance $x = \pm 20$ from the cluster center is not considered, since the cross-sectional area in that point is zero.

2.4 Computing the Energy

In order to test whether the energy is conserved, the initial energy of the system can be calculated and printed together with the final energy after a specific time interval. According to the conservation of energy, these are equal.

The total energy E_{tot} of the system is found by summing up the potential energy E_{pot} and kinetic energy E_{kin} of the N bodies that constitutes the system. The total potential energy is calculated as

$$E_{pot} = \sum_{1=0}^{N} \sum_{j \neq i} \frac{m_i m_j}{r_{ij}} \tag{2.19}$$

in which m_i and m_j are the masses of the i'th and j'th body, respectively, $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ is the distance between the two bodies, and G is the gravitational constant. The total kinetic energy of the system is calculated as

$$E_{kin} = \frac{1}{2} \sum_{1=0}^{N} m_i v_i^2 \tag{2.20}$$

with v_i being the speed of the i'th particle calculated as $v_i = \sqrt{v_{ix}^2 + v_{iy}^2 + v_{iz}^2}$, and m_i is the corresponding mass of that body.

The c++ code for computing the total energy of the system is given here below. When the kinetic energy is calculated, v_i is not explicitly calculated. Instead v_i^2 is calculated to reduce the number of floating point operations.

```
for (int i=0; i<number_of_particles; i++){
   for (int k=0; k<3; k++){
      kin_en(i) += v(i,k)*v(i,k);
    }
    kin_en(i) = 0.5*m(i)*kin_en(i);
    for (int j=0; j<number_of_particles; j++){
        if (j != i){
         pot_en(i) += pow(distance_between_particles(i,j),-1.0)*m(j);
        }
    }
    pot_en(i) = pot_en(i)*G*m(i);
    tot_en(i) = kin_en(i)+pot_en(i);
}
```

CHAPTER

3 RESULTS AND DISCUSSION

The results from running the codes described in Chap. 2 for computing the blah blah blah ?? can be found in the GitHub folder `https:/??`, together with the MatLab scripts for the plots presented in this chapter.[1]

3.1 Stability of the 2-body System using Runge-Kutta and Velovity-Verlet

Table 3.1. Initial position and velocity for Earth and Sun in the Sun-Earth-like two-body system. F1 refers to the frame of reference at which the Sun is at rest in origo at all times, whilst F2 refers to the frame of reference in which both the Sun and the Earth moves relative to the coordinate axis. The mass of the Earth is given in solar masses, that is $M_E = 3.0 \times 10^{-6} M_\odot$, and the gravitational constant is given is $2.96 \cdot 10^{-4} \frac{\mathrm{AU}^2}{\mathrm{days}^2 \mathrm{M}_\odot}$ (see Sec. 2.1).

	$\mathbf{r}_{initial}$ [AU]	$\mathbf{v}_{initial}$ [AU/day]
Earth (F1)	$(1.0, 0.0, 0.0)$	$(0.0, 0.017, 0.0)$
Sun (F2)	$(1.0, 1.0, 1.0)$	$(0.0, 0.0, 0.0)$
Earth (F2)	$(2.0, 1.0, 1.0)$	$(0.0, 0.017, 0.0)$

[1] FiXme Note: fix these lines

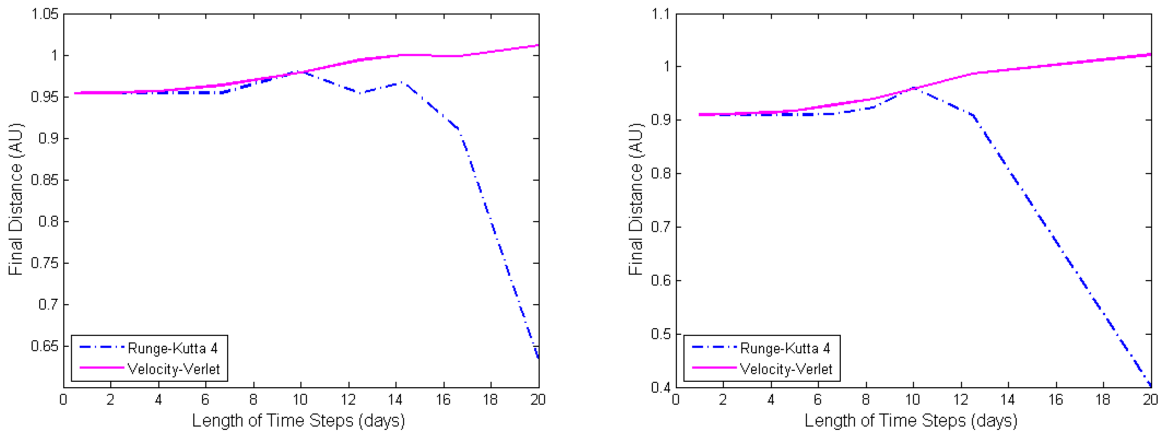


Figure 3.1. Distance between bodies after 100 years as a function of time step length for the Earth-Sun-like two-body system using both the forth order Runge-Kutta method and the Velocity-Verlet method. The leftmost plot do not allow for Sun motion relative to the frame of reference, whilst the rightmost allows for movement of both the Earth and the Sun relative to the frame of reference.

Comment here on, when the methods become stable: // around step length of 8 days for RK4 and 4 days for VV ?? [2]

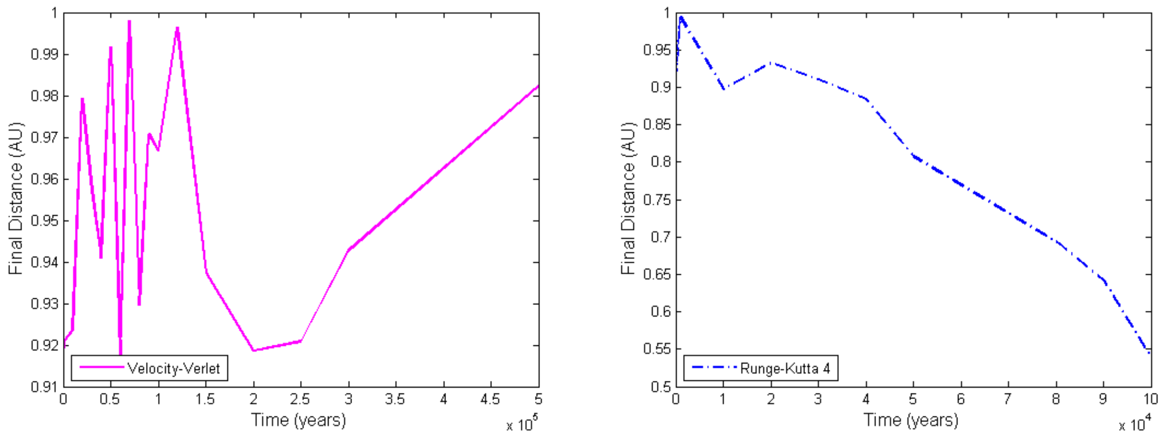


Figure 3.2. The final distance as a function of time with a time step length of 5 days for both the Velocity-Verlet and Runge-Kutta method with both the Earth and Sun moving relative to the frame of reference. After 2.5×10^4 years, the Earth continuously moves towards the Sun, in the Runge-Kutta method, whilst the distance between the Earth and Sun still fluctuates between 0.92 AU and 1 AU for the Velocity-Verlet method after 5×10^5 years.

Table 3.2. Computational time for the fourth order Runge-Kutta method and the Velocity-Verlet method for different time steps during 1 year.

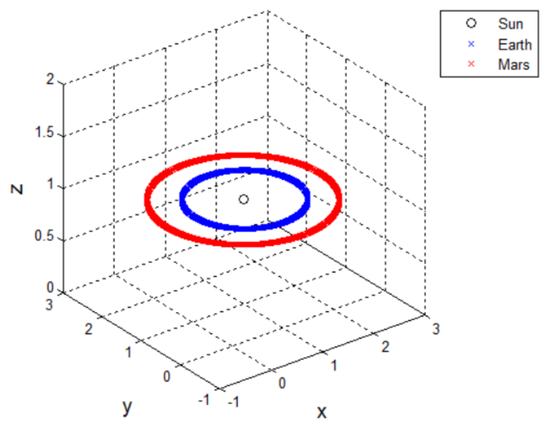
# time steps	Comp. time RK4	Comp. time VV
1	6	2
10	25	8
10^4	8.4×10^3	4.6×10^3
10^6	8.0×10^5	3.6×10^5

[2]FiXme Note: fix these lines

3.2 Testing Runge-Kutta and Velocity-Verlet for Sun-Earth-Mars System

Table 3.3. Mass, initial position and initial velocity of Sun, Earth and Mars when running the Runge-Kutta 4 algorithm for this three-body problem. The Earth and Mars are set to orbit in the $x-y$-plane at $z=1$ AU with the distance 1 AU and 1.5 AU to the Sun, respectively, which is not physically true. However, this initialization of position and velocity is reasonable to illustrate the validity of the Runge-Kutta method and Velocity-Verlet method presented in [3].

	mass $[M_\odot]$	$\mathbf{r}_{initial}$ [AU]	$\mathbf{v}_{initial}$ [AU/day]
Sun	1.0	$(1.0, 1.0, 1.0)$	$(0.0, 0.0, 0.0)$
Earth	3.0×10^{-6}	$(2.0, 1.0, 1.0)$	$(0.0, 0.017, 0.0)$
Mars	3.2×10^{-7}	$(-0.5, 1.0, 1.0)$	$(0.0, 0.014, 0.0)$



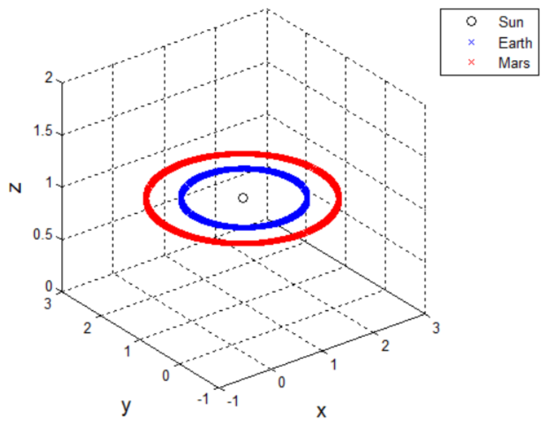


Figure 3.3. Time evolution of the simplified system of Sun-Earth-Mars over a time period of 20 years using Runge-Kutta (leftmost) and Velocity-Verlet (rightmost) method with a time step length of 1 day. The masses, initial positions, and initial velocities of the three objects are given in Tab. 3.3.

CHAPTER

4

CONCLUSION

Bibliography