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: 15
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 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.

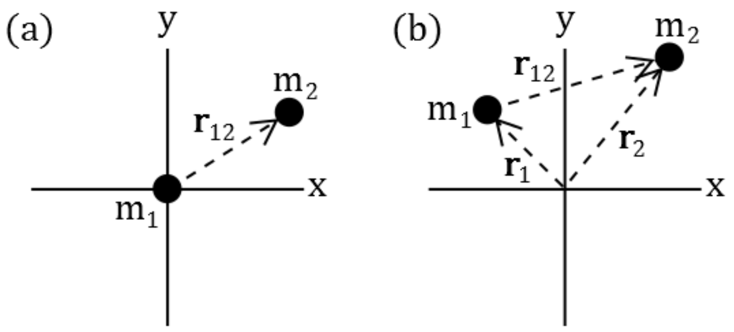

Figure 2.1. Two-dimensional illustration of the three-dimensional problem of determining determining the relative distance and relative velocity between two bodies. In (a) body 1 with mass m_1 is considered stationary in position- and velocity-space, whilst body 2 with mass m_2 moves relative to body 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.

[1] FiXme Note: fix these lines

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

$$\frac{dy}{dt} = f(t,y) \tag{2.1}$$

Which yields that

$$y(t) = \int f(t,y)dt \tag{2.2}$$

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

$$\frac{d\mathbf{r}}{dt} = \mathbf{v} \qquad \text{and} \qquad \frac{d\mathbf{v}}{dt} = -\frac{GM_1M_2}{r^3}\mathbf{r} \tag{2.3}$$

[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 (2.3) 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];
}
```

2.1.1 Velocity-Verlet method

- Remember to write about accuracy of algorithm!!

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

$$\mathbf{r}(t_i \pm \delta t) = \mathbf{r}(t_i) \pm \mathbf{v}(t_i)\delta t + \mathbf{a}(t_i)\frac{\delta t^2}{2} \pm \frac{\delta t^3}{6}\frac{d^3\mathbf{r}(t_i)}{dt^3} + \mathscr{O}(\delta t^4) \tag{2.4}$$

Adding the two expressions in Eq. (2.4) gives

$$\mathbf{r}(t_i + \delta t) = 2\mathbf{r}(t_i) - \mathbf{r}(t_i - \delta t) + \mathbf{a}(t_i)\delta t^2 + \mathscr{O}(\delta t^4) \tag{2.5}$$

[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

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

$$\mathbf{r}(t_i+\delta t)=\mathbf{r}(t_i)+\mathbf{v}(t_i)\delta t+\frac{1}{2}\mathbf{a}(t_i)\delta t^2 \tag{2.6}$$

$$\mathbf{v}(t+\delta t)=\mathbf{v}(t)+\frac{1}{2}(\mathbf{a}(t)+\mathbf{a}(t+\delta t))\delta t \tag{2.7}$$

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

$$\mathbf{v}_{part1}(t+\delta t)=\mathbf{v}(t)+\frac{1}{2}\mathbf{a}(t)\delta t \tag{2.8}$$

and then use ?? [6] to determine $\mathbf{a}(t+\delta t)$, which is then used to compute the remaining term of Eq. (2.7) as

$$\mathbf{v}_{part2}(t+\delta t)=\frac{1}{2}\mathbf{a}(t+\delta t)\delta t \tag{2.9}$$

The velocity-Verlet method uses the algortihm *Derivative* described in Sec. 2.1, 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;
}
```

2.1.2 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}.

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

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

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{2.10}$$

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. 2.1 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. (2.10), 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;
    }
```

2.2 Generating Position, Mass and Velocity for Cluster Particles

[7] [8]

2.2.1 Gaussian Distributed Mass

[9]

```
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{
      v1 = 2.*((double) rand() / (RAND_MAX)) -1.0;
      v2 = 2.*((double) rand() / (RAND_MAX)) -1.0;
      rsq = v1*v1+v2*v2;
    } while (rsq >= 1.0 || rsq == 0.);
    fac = sqrt(-2.*log(rsq)/rsq);
    gset = v1*fac;
    iset = 1;
    mass(i) = v2*fac;
    mass(i) += 10;
  }
}
```

[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: write here, what kind of distribution, we want!

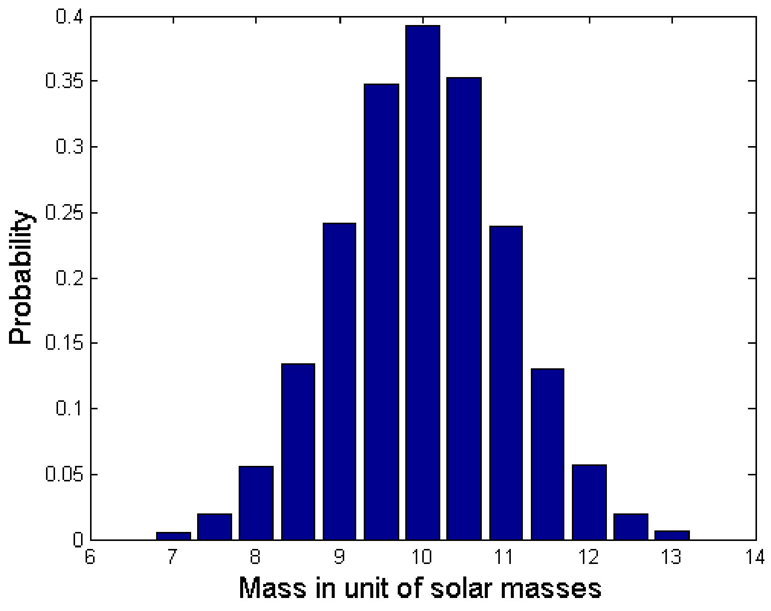

Figure 2.2. Histogram of the mass of 100,000 particles generated by the c++ code introduced [10].

11

2.2.2 Uniformly Distributed Position

12

```
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));
   }
```

[11]FiXme Note: eq. include gaussian dist. in fig
[12]FiXme Note: write here, what kind of distribution, we want!

```
}
```

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. 2.4 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. **??**).

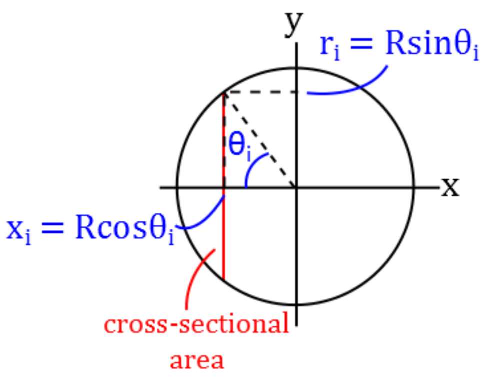

Figure 2.3. Two-dimensional illustration of the three-dimensional problem of determining the density of particles in each x-value.

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 $r_i = 20 sin\theta_i$ ly. This yields that the area A_i of the cross-sectional area, in ly, is given as

$$A_i = 400\pi sin^2\theta_i = 400\pi(1 - cos^2\theta) \tag{2.11}$$

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

$$A_i = \pi(400 - x_i^2) \tag{2.12}$$

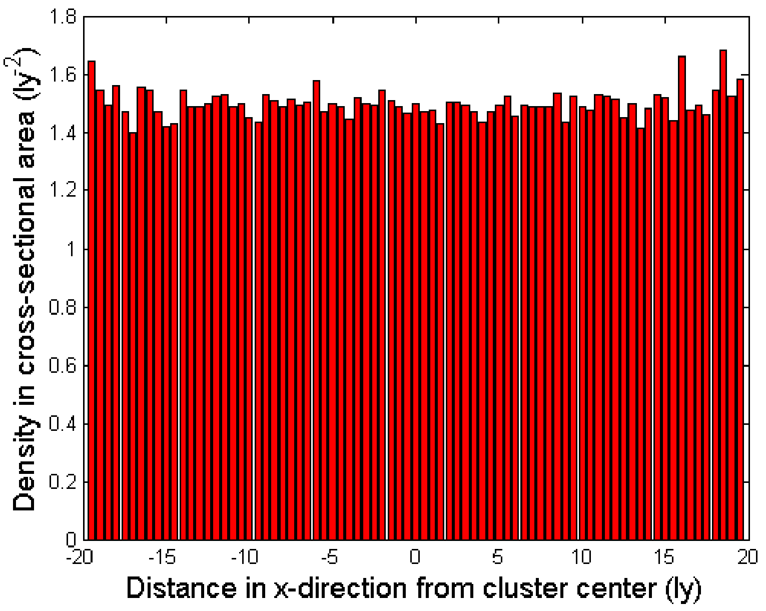

Figure 2.4. Histogram of density of 100,000 particles with position generated by the code introduced in [13] as a function of the x-coordinate of the particles. The histogram is made with bins in the interval $[-19.5; 19.5]$ and a bin-size of 0.5. The distance $x = \pm 20$ from the cluster center is not considered, since the cross-sectional area in that point is zero.

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

Table 3.1. 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 [2].

	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}	$(-0.5, 1.0, 1.0)$	$(0.0, 0.014, 0.0)$

[1] FiXme Note: fix these lines

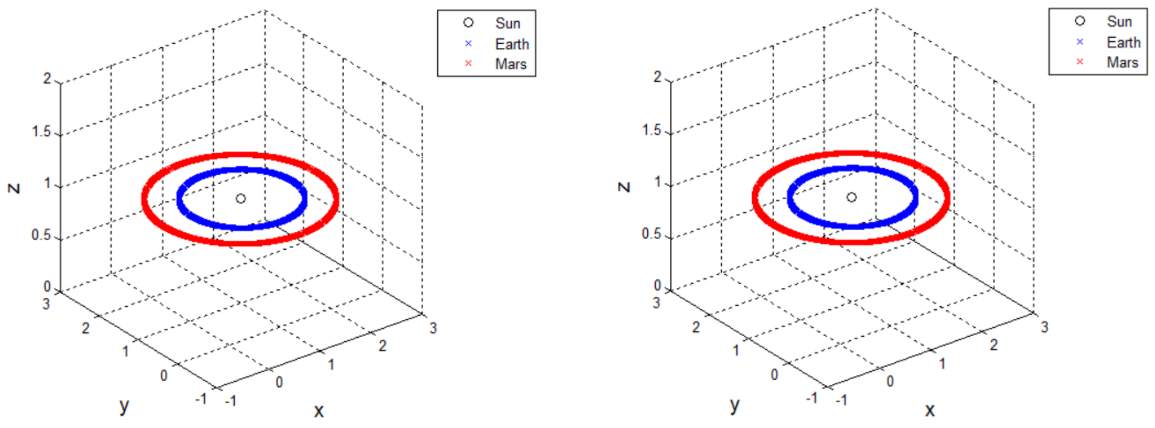

Figure 3.1. Time evolution of the simplified system of Sun-Earth-Mars over a time period of 20 years using Runge-Kutta (leftmost) and Velocity-Verlet (rightmost) method with a time step length of 1 day. The masses, initial positions, and initial velocities of the three objects are given in Tab. 3.1.

CHAPTER

4

CONCLUSION

BIBLIOGRAPHY