FYS4150 - Computational Physics - Project 3

Eimund Smestad
Email: eimundsm@fys.uio.no
25th October, 2014

Abstract

In this project I have simulated the solar system by solving the Newton's law of gravitation with the Verlet and RK4 method for second order ODE's. Computation time, accuracy and stability are considered.

1 Newton's law of gravitation

Newton's law of gravitation force F between two bodies of masses m and M with a distance r between them are given by

$$F = G\frac{mM}{r^2} \tag{1}$$

where $G = 6.67384 \times 10^{-11}$ [m^3 / $\mathrm{kg} \cdot \mathrm{s}^2$] is the gravitational constant. The gravitational force acts in radial direction between the bodies towards each other. Assuming the bodies are coplanar and using Newton's second law of motion $F = ma$ in Cartesian coordinates $r^2 = \sum_{i=1}^{2} {x_i}^2$ we can write

$$\frac{\mathrm{d}^2 x_i}{\mathrm{d}t^2} = \frac{F_i}{m} \qquad \text{for } i \in \mathbb{N}_1^2$$

where the subscript i indicates the Cartesian coordinate number for body m. This can be written as set of coupled first order differential equations

$$\frac{\mathrm{d}v_i}{\mathrm{d}t} = \frac{F_i}{m} \qquad \text{and} \qquad \frac{\mathrm{d}x_i}{\mathrm{d}t} = v_i \qquad \text{for } i \in \mathbb{N}_1^2 .$$

We need to have initial conditions for the bodies, and we should know the initial position. But we also need an initial velocity as well to define the system. If we assume that the body m goes around M we can guess the initial velocity with the centripetal force

$$F = \frac{mv^2}{r} ,$$

which applies for circular motion. Combining the centripetal force with Newton's law of gravitation (1) we can now calculate the velocity

$$v = \pm\sqrt{\frac{GM}{r}} ,$$

where we need to decide direction of the tangential velocity v. From this we can find the initial velocity.

If we have many bodies that interact with gravitational force on each other, then the force $\vec{F}_i$ that acts on the body i adds the the gravitational force $\vec{F}_{ij}$ from the other bodies j as follows

$$\vec{F}_i = \sum_j \vec{F}_{ij} \,.$$

The radial escape velocity v_e of m from M can be derived from (1) by using Newton's second law $F = ma$, which gives us the following equation to start with

$$\frac{\mathrm{d}^2 r}{\mathrm{d}t^2} = \frac{\mathrm{d}v_r}{\mathrm{d}t} = \frac{\mathrm{d}v_r}{\mathrm{d}r}\frac{\mathrm{d}r}{\mathrm{d}t} = v_r \frac{\mathrm{d}v_r}{\mathrm{d}r} = -G\frac{M}{r^2} \,,$$

where the minus sign comes from the fact that the force acts in opposite direction of motion. We can integrate up this expression as follows

$$\int_{v_0}^{v_r} v_r \,\mathrm{d}v_r = -GM \int_{r_0}^{r} \frac{\mathrm{d}r}{r^2} \,,$$

which leads to the radial velocity

$$v_r = \sqrt{{v_0}^2 + 2GM\left(\frac{1}{r} - \frac{1}{r_0}\right)} .$$

The escape velocity v_e is given when the expression in the square root is zero and $r \to \infty$;

$$\lim_{r\to\infty} {v_0}^2 + 2GM\left(\frac{1}{r} - \frac{1}{r_0}\right) = 0 \,,$$

which gives

$$v_e = \sqrt{\frac{2GM}{r_0}} \,. \tag{2}$$

Note that the escape velocity v_e is independent of the mass m of the escaping body, but depends on the mass M of the body that it tries to escape.

2 Numerical methods

In this section we will discretize the following equation set

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v\,(x,t) \qquad \text{and} \qquad \frac{\mathrm{d}v}{\mathrm{d}t} = a\,(x,t) \,. \tag{3}$$

2.1 Verlet algorithm

Using Taylor expansion of $x(t)$ with a step size Δt forward and backwards yields

$$x(t \pm \Delta t) = \sum_{i=0}^{\infty} \frac{\overset{(i)}{x}(t)}{i!} (\pm \Delta t)^i \,, \tag{4}$$

where $\frac{\mathrm{d}^i x}{\mathrm{d}t^i} = \overset{(i)}{x}(t)$. Adding the forward and backward expansion yields

$$x(t + \Delta t) + x(t - \Delta t) = 2 \sum_{i=0}^{\infty} \frac{\overset{(2i)}{x}(t)}{(2i)!} \Delta t^{2i} \,.$$

This can be discretized as

$$x_{i+1} = 2x_i - x_{i-1} + a_i \Delta t^2 \qquad \text{for } i \in \mathbb{N}_0 \text{ and given } x_0 \text{ and } x_{-1}, \tag{5}$$

which has a local truncation error $O\left(\Delta t^4\right)$. Usually we have the initial condition x_0, however we might have initial velocity v_0 instead of x_{-1}. To resolve this we may us the backward Euler scheme

$$\frac{x_0 - x_{-1}}{\Delta t} = \overset{(1)}{x_0} = v_0 \,,$$

which yields

$$x_{-1} = x_0 - v_0 \Delta t \,,$$

and the initial step is then given by

$$x_1 = x_0 + v_0 \Delta t + a_0 \Delta t^2 \tag{6}$$

The discretization of the velocity v can be done by first subtracting the forward and backward expansion in (4):

$$x(t + \Delta t) - x(t - \Delta t) = 2 \sum_{i=0}^{\infty} \frac{\overset{(2i+1)}{x}}{(2i+1)!} \Delta t^{2i+1}$$

which leads to

$$v_i = \frac{x_{i+1} - x_{i-1}}{2\Delta t} \tag{7}$$

with local truncation error $O\left(\Delta x^2\right)$. We face a similar problem when calculating the last point of velocity v_{n-1} as we did for the initial step of x_1, I resolve this by using the following forward Euler scheme

$$v_{n-1} = v_{n-2} + a_{n-2} \Delta t \,. \tag{8}$$

2.2 Second order ODE with RK4

We start by do polynomial interpolation with Lagrange polynomials of the points (t_i, a_i) with $i \in \mathbb{N}_0^2$;

$$p(t) = \sum_{i=0}^{2} a_i \prod_{\substack{0 \le j \le n \\ j \neq i}}^{n} \frac{t - t_j}{t_i - t_j},$$

where $a_i = a(t_i, x_i)$. Integrating this polynomial with the endpoints t_0 and t_2 ans choosing $t_1 = \frac{t_0+t_2}{2}$ as the midpoint, we get;

$$\int_{t_0}^{t_2} p(t)\ \mathrm{d}t = \frac{t_2 - t_0}{6}\left(a_0 + 4a_1 + a_2\right).$$

When we discretize this integral we get the velocity steps

$$v_{i+1} = v_i + \frac{\Delta t}{6}\left(a_i + 4a_{i+1/2} + a_{i+1}\right),$$

with $O\left(\Delta t^5\right)$ truncation error. We can approximate this difference equation with the so called RK4 method

$$v_{i+1} = v_i + \frac{\Delta t}{6}\left(a_i + 2k_{v1} + 2k_{v2} + k_{v3}\right) \tag{9}$$

$$k_{v1} = a\left(t_i + \frac{\Delta t}{2}, x_i + \frac{\Delta t}{2}a_i\right) \tag{10}$$

$$k_{v2} = a\left(t_i + \frac{\Delta t}{2}, x_i + \frac{\Delta t}{2}k_{v1}\right) \tag{11}$$

$$k_{v3} = a\left(t_i + \Delta t, x_i + \Delta t k_{v2}\right). \tag{12}$$

We can do similar discretization with for position steps

$$x_{i+1} = x_i + \frac{\Delta t}{6}\left(v_i + 4v_{i+1/2} + v_{i+1}\right),$$

and the RK4 approximation yields

$$x_{i+1} = x_i + \frac{\Delta t}{6}\left(v_i + 2k_{x1} + 2k_{x2} + k_{x3}\right) \tag{13}$$

$$k_{x1} = v_i + \frac{\Delta t}{2}a_i \tag{14}$$

$$k_{x2} = v_i + \frac{\Delta t}{2}k_{v1} \tag{15}$$

$$k_{x3} = v_i + \Delta t k_{v2}. \tag{16}$$

3 Numerical implementation

I have made a `template<class T> class Differential_2` that contains both the Verlet and RK4 solver. The solvers are accessed through the `Differential_2::Solve`

`Differential_2::Solve`

```
public: template<DifferentialType Type, class C> void Solve(C* owner, T* (C
    ::*f)(T*), T* t, T** x0, T** v0, T dt, unsigned int dim, unsigned int
    step) {
        /* owner - the owner of the data
        * f - callback function to calculate forces
        * t - array to the time steps
        * x0 - array to all the position components for all the bodies for
             every time step. Initial value should be set at x0[i][0] prior
             to call
        * v0 - array for all the velocity components for all the bodies
             for every time step. Initial value should be set at x0[i][0]
             prior to call
        * dim - number of vector components times number of bodies (width
             of the arrays)
        * step - number of time steps (length of the arrays)
        */

        t[0] = 0;
        t[1] = dt;
        Diff<Type,C>::Solve(owner, f, t, x0, v0, dt, dim, step);
}
```

This function takes a function pointer as an argument to able to execute force calculation inside the Verlet and RK4 solver. This is a generic implementation of the solvers and can execute any type of calculation, instead of force calculation. Note that all the bodies are solve at once inside these solvers. To tell which solver you want to use, you send is with the template argument `DifferentialType`.

`enum DifferentialType`

```
enum class DifferentialType {
        RK4,
        Verlet
};
```

The `Differential_2::Solve` is partially specialized by `template<DifferentialType Type, class C> class Diff` to give the different solvers.

`Differential_2::Diff<DifferentialType::Verlet,C>::Solve`

```
inline static void Solve(C* owner, T* (C::*f)(T*), T* t, T** x0, T** v0, T
    dt, unsigned int dim, unsigned int step) {
        T* a;
        T* x2 = new T[dim];

        // Initial step
        for(unsigned int j = 0; j < dim; j++)
                x2[j] = x0[j][0];
        a = (owner->*f)(x2);
        for(unsigned int j = 0; j < dim; j++)
                x0[j][1] = x2[j] + dt*(v0[j][0] + dt*a[j]);      // eq 6
```

```
        // Continuing step
        for(unsigned int i = 2, i1=1, i2=0, j; i < step; i++, i1++, i2++) {
                for(j = 0; j < dim; j++)
                        x2[j] = x0[j][i1];
                a = (owner->*f)(x2);
                for(j = 0; j < dim; j++) {
                        x0[j][i] = 2*x2[j] - x0[j][i2] + dt*dt*a[j];// eq 5
                        v0[j][i1] = (x0[j][i]-x0[j][i2])/(2*dt); // eq 7
                }
                t[i] = t[i1]+dt;
        }

        // Last step
        unsigned int i1 = step-1;
        unsigned int i2 = step-2;
        for(unsigned int j = 0; j < dim; j++)
                v0[j][i1] = v0[j][i2] + a[j]*dt;         // eq 8
        delete [] x2;
}
```

Differential_2::Diff<DifferentialType::RK4,C>::Solve

```
inline static void Solve(C* owner, T* (C::*f)(T*), T* t, T** x0, T** v0, T
    dt, unsigned int dim, unsigned int step) {
        T* a;
        T* x = new T[dim];
        T* x2 = new T[dim];
        T* v = new T[dim];
        T* v2 = new T[dim];
        T dt2 = dt/2;
        T dt6 = dt/6;

        for(unsigned int i = 1, i1 = 0, j; i < step; i++, i1++) {
                for(j = 0; j < dim; j++) {
                        x[j] = x0[j][i1];
                        v[j] = v0[j][i1];                // v_i
                }
                a = (owner->*f)(x);                      // a_i
                for(j = 0; j < dim; j++) {
                        x0[j][i] = v[j];
                        v0[j][i] = a[j];
                        x2[j] = x[j] + v[j]*dt2;
                        v2[j] = v[j] + a[j]*dt2;         // k_x1 (eq 14)
                }
                a = (owner->*f)(x2);                     // k_v1 (eq 10)
                for(j = 0; j < dim; j++) {
                        x0[j][i] += 2*v2[j];
                        v0[j][i] += 2*a[j];
                        x2[j] = x[j] + v2[j]*dt2;
                        v2[j] = v[j] + a[j]*dt2;         // k_x2 (eq 15)
                }
                a = (owner->*f)(x2);                     // k_v2 (eq 11)
                for(j = 0; j < dim; j++) {
                        x0[j][i] += 2*v2[j];
                        v0[j][i] += 2*a[j];
                        x2[j] = x[j] + v2[j]*dt;
```

```
                        v2[j] = v[j] + a[j]*dt;          //k_x3 (eq 16)
                }
                a = (owner->*f)(x2);                     // k_v3 (eq 12)
                for(j = 0; j < dim; j++) {
                        x0[j][i] += v2[j];
                        x0[j][i] *= dt6;
                        x0[j][i] += x[j];                // x_(i+1) (eq 13)
                        v0[j][i] += a[j];
                        v0[j][i] *= dt6;
                        v0[j][i] += v[j];                // v_(i+1) (eq 9)
                }
                t[i] = t[i1]+dt;
        }
        delete [] x;
        delete [] v;
        delete [] x2;
        delete [] v2;
}
```

This project is about solving the solar system, and therefore I made `template<class T, unsigned int DIM> class System : Differential_2<T>` which is a class which contains a collection of bodies that interact with each other through gravitation, (1). The `System` class has been prepared to support arbitrary number of spatial vector components through the template argument `DIM`. The `System` class inherits from `Differential_2<T>` which enables it to use the Verlet or RK4 solver to simulate the solar system.

`System::Run`

```
template<DifferentialType Type> void Run(T t, unsigned int n) {
        T dt = t/(n-1);
        length = n;
        this->template Solve<Type>(this, &System<T,DIM>::Gravity, this->t,
            x, v, dt, width, n);
}
```

We see that `System::Gravity` is used as callback function to Verlet or RK4 solvers, which calculates acceleration of all the bodies at once.

`System::Gravity`

```
T* Gravity(T* x) {
        T d,r, diff[DIM];
        auto b1 = body->next;
        decltype(b1) b2;
        for(unsigned int i = 0; i < width; i++)
                a[i] = 0;        // Reset force

        for(unsigned int i = 0, j, k, i1, j1; i < width; i+=DIM, b1=b1->
            next) {
                for(j=i+DIM, b2=b1->next; j < width; j+=DIM, b2=b2->next) {
                        r = 0;

                        // Calculate distance between two bodies
                        for(k=0, i1=i, j1=j; k < DIM; k++, i1++, j1++) {
                                diff[k] = x[i1]-x[j1];
                                r += diff[k]*diff[k];
                        }
```

```
                        if(r) {
                                r = G/(r*sqrt(r));
                                // Calculate the acceleration component k
                                    for both bodies
                                for(k=0, i1=i, j1=j; k < DIM; k++, i1++, j1
                                    ++) {
                                        d = r*diff[k];
                                        a[i1] -= d*b2->element->m;
                                        a[j1] += d*b1->element->m;
                                }
                        }
                }
        }
        return a;
}
```

The planets are stored in an costume made array `Array<Body<T,DIM>*>* body` which does not need to reallocate memory when a new element is added. Each planet have an object to `template<class T, unsigned int DIM> struct Body` to represent it self. Array contains the array element and pointer to the previous and next element. This pointers enables just to insert a new element to the array by just rearranging the neighboring elements pointers.

struct Array

```
template<class T> struct Array {
        T element;
        Array<T>* prev;
        Array<T>* next;
        Array() {
                prev = this;
                next = this;
        }
        ~Array() {
                if(next != this)
                        delete next;
        }
        T operator[] (const int i) {
                if(prev != this) {
                        if(i) {
                                if(i > 0)
                                        return (*next)[i-1];
                                else
                                        return (*prev)[i+1];
                        }
                        return this->element;
                } else if(next != this)
                        return (*next)[i];
                return (T)Null<T>::value;
        }
        void Add(T element) {
                if(next != this)
                        next->Add(element);
                else {
                        next = new Array<T>;
                        next->prev = this;
                        next->next = next;
```

```
                        next->element = element;
                }
        }
        int Length() {
                if(next != this)
                        return next->Length()+1;
                return 0;
        }
};
```

4 Result

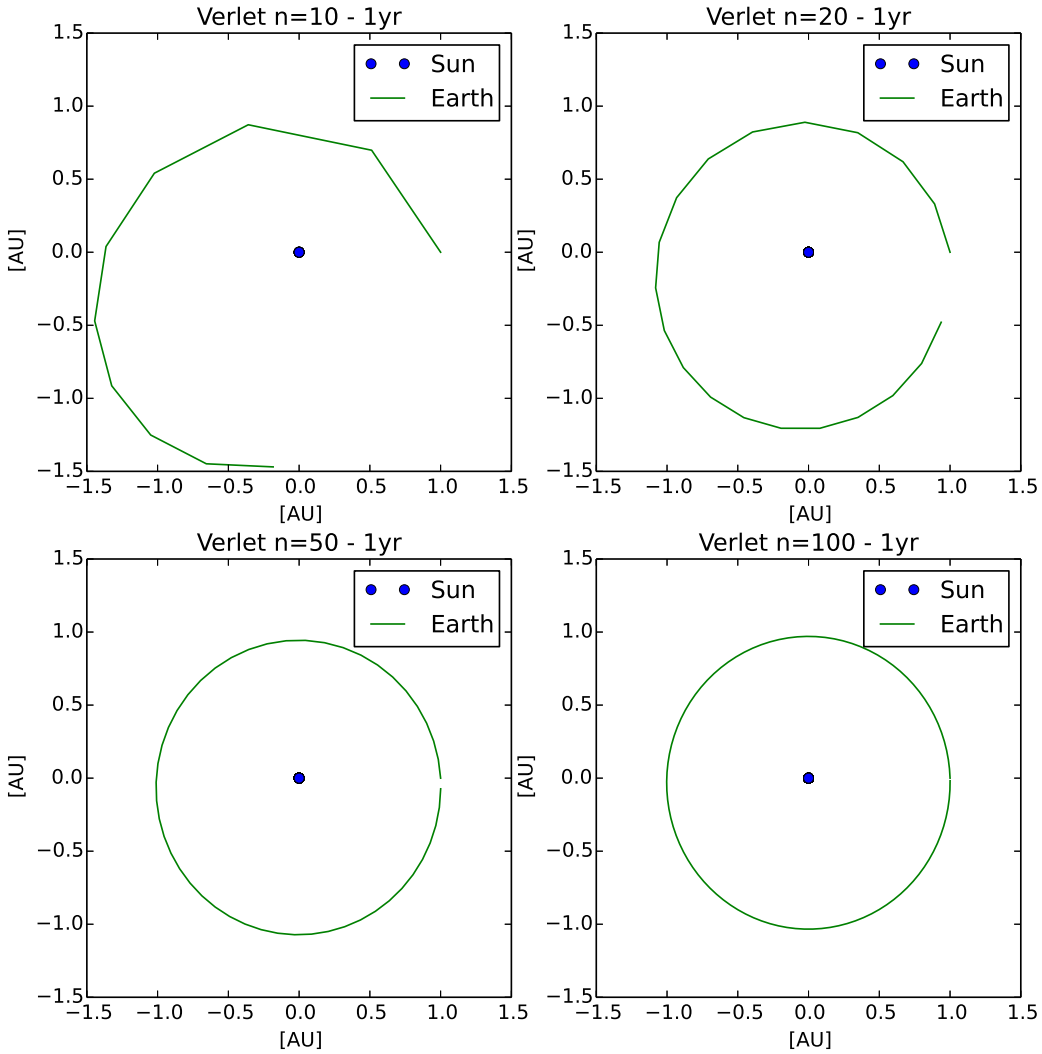

Figure 4.1: *Simulation of the orbit for the earth and sun system only, where the Verlet solver is used for different number of steps n. The initial velocity for the earth is assumed from the centripetal acceleration, and the sun's initial velocity is zero.*

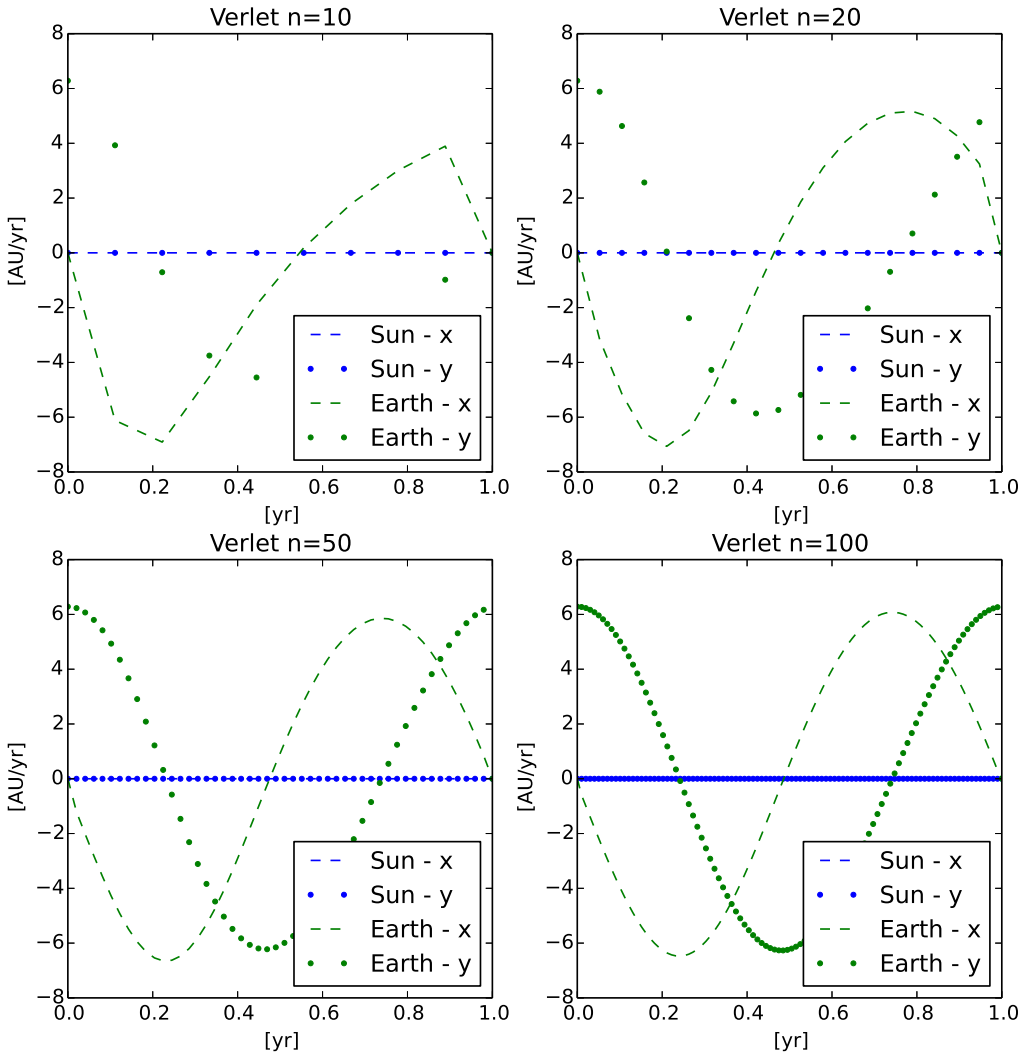

Figure 4.2: *Simulation of the periodicity for the earth and sun system only, where the Verlet solver is used for different number of steps n. The initial velocity for the earth is assumed from the centripetal acceleration, and the sun's initial velocity is zero.*

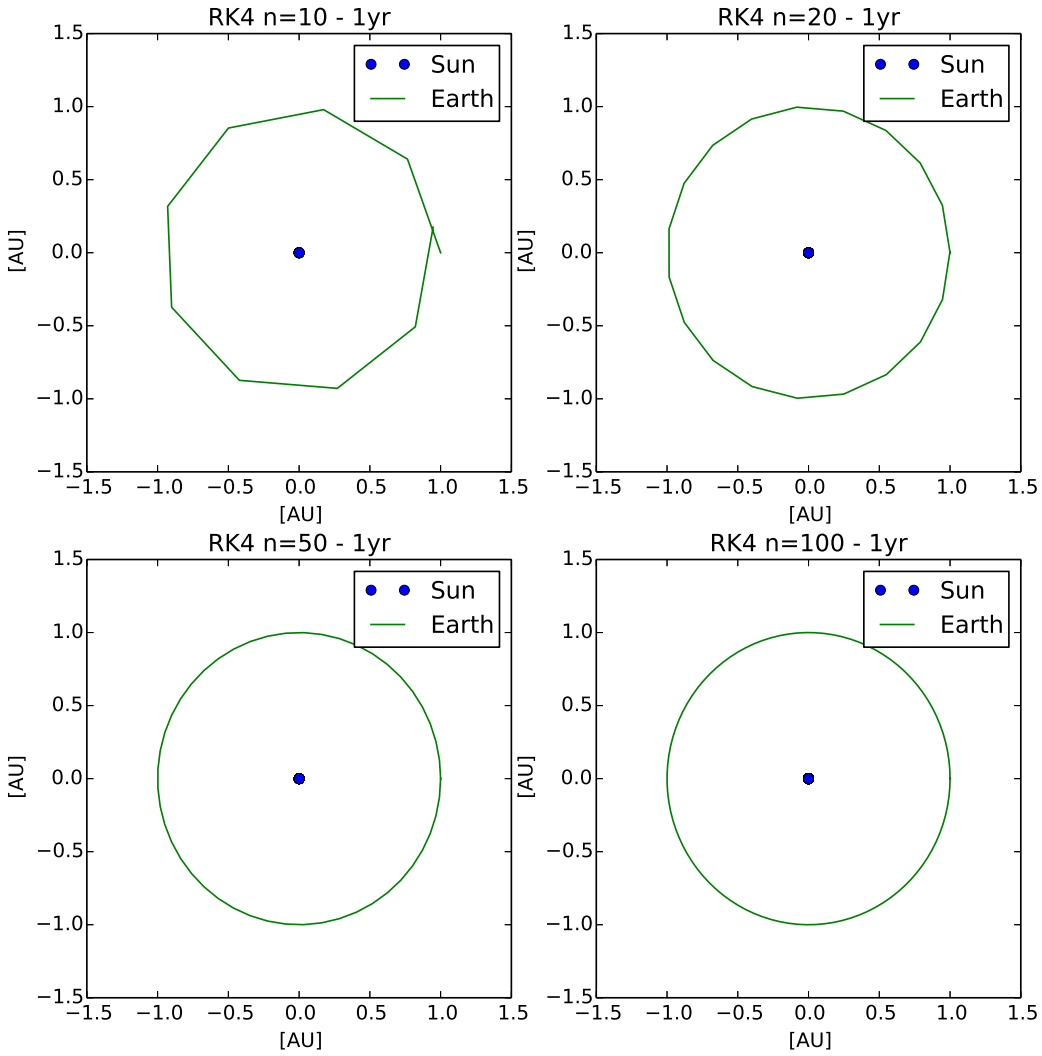

Figure 4.3: *Simulation of the orbit for the earth and sun system only, where the RK4 solver is used for different number of steps n. The initial velocity for the earth is assumed from the centripetal acceleration, and the sun's initial velocity is zero.*

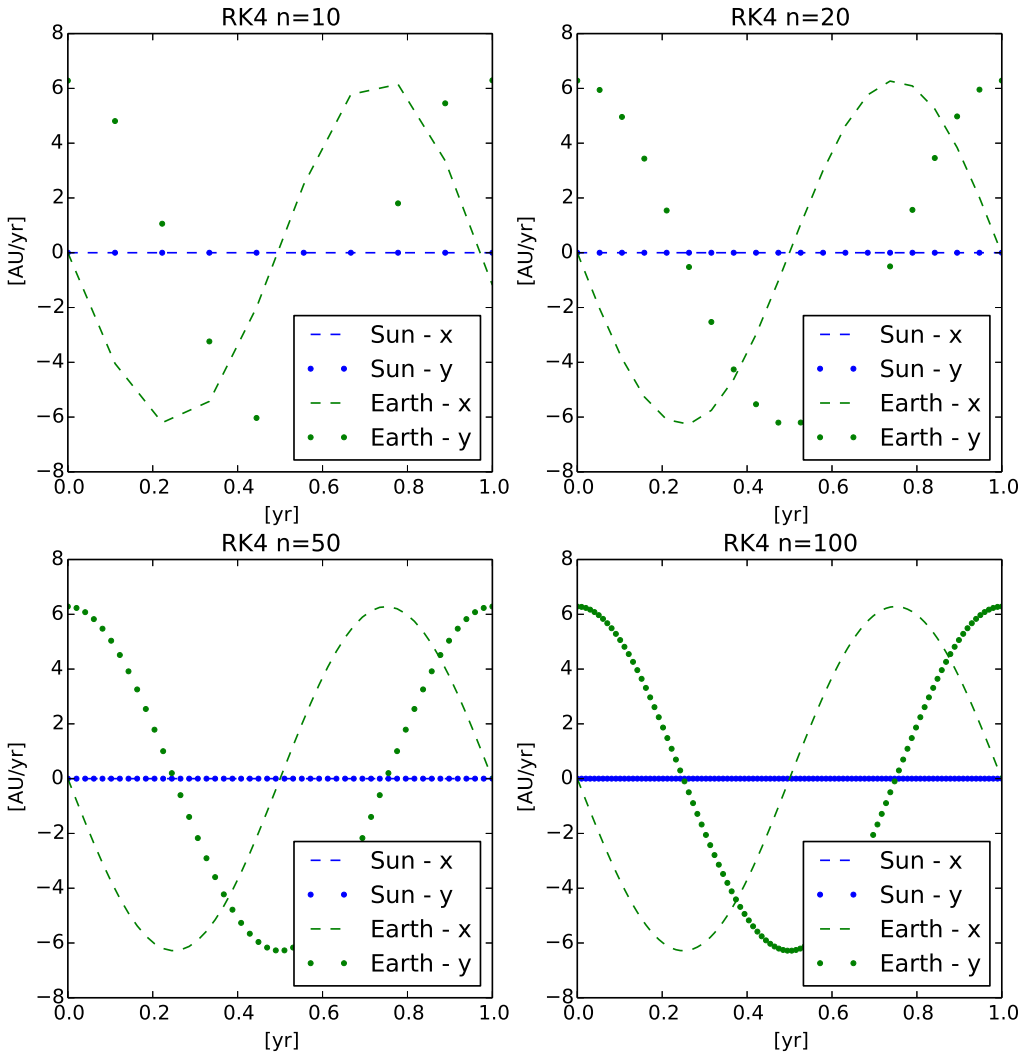

Figure 4.4: *Simulation of the periodicity for the earth and sun system only, where the RK4 solver is used for different number of steps n. The initial velocity for the earth is assumed from the centripetal acceleration, and the sun's initial velocity is zero.*

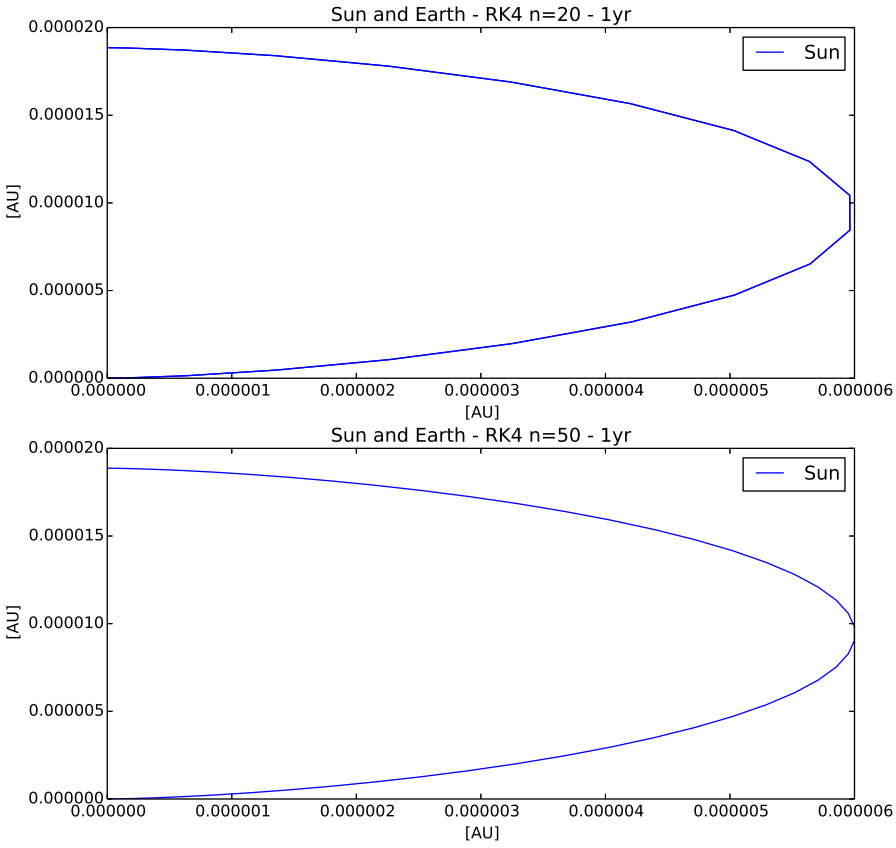

Figure 4.5: *Simulation of the sun's movment for the earth and sun system only, where the RK4 solver is used for different number of steps n. The initial velocity for the earth is assumed from the centripetal acceleration, and the sun's initial velocity is zero. The orbit of the sun does not return to initial position because the initial momentum for the earth and sun system is not zero.*

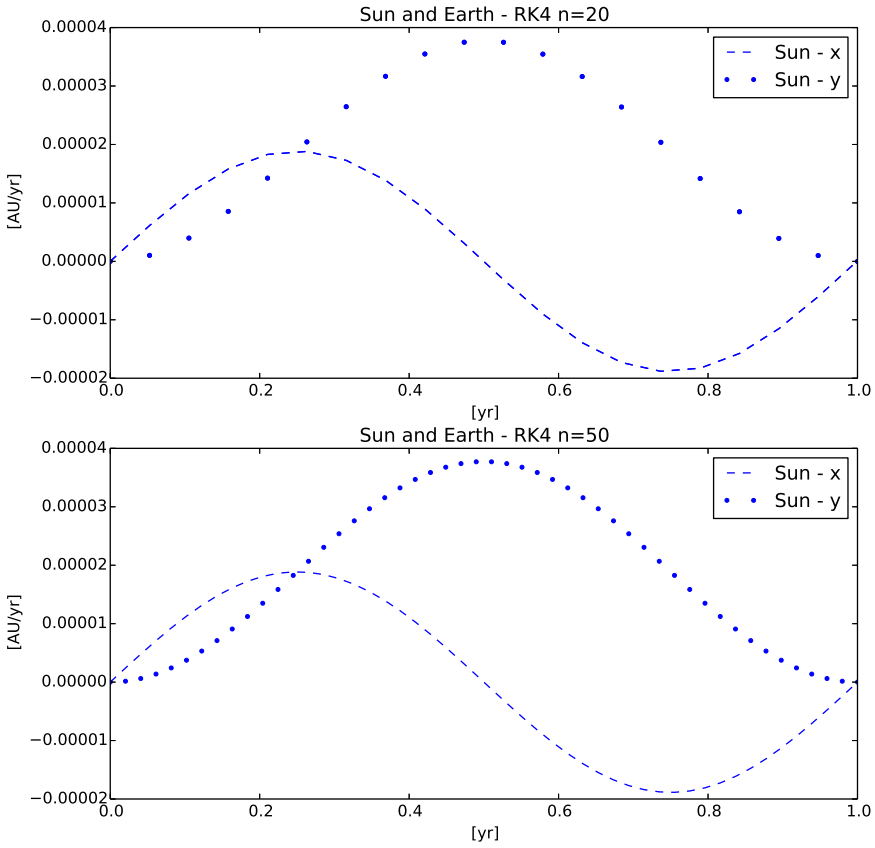

Figure 4.6: *Simulation of the sun's periodicity for the earth and sun system only, where the RK4 solver is used for different number of steps n. The initial velocity for the earth is assumed from the centripetal acceleration, and the sun's initial velocity is zero.*

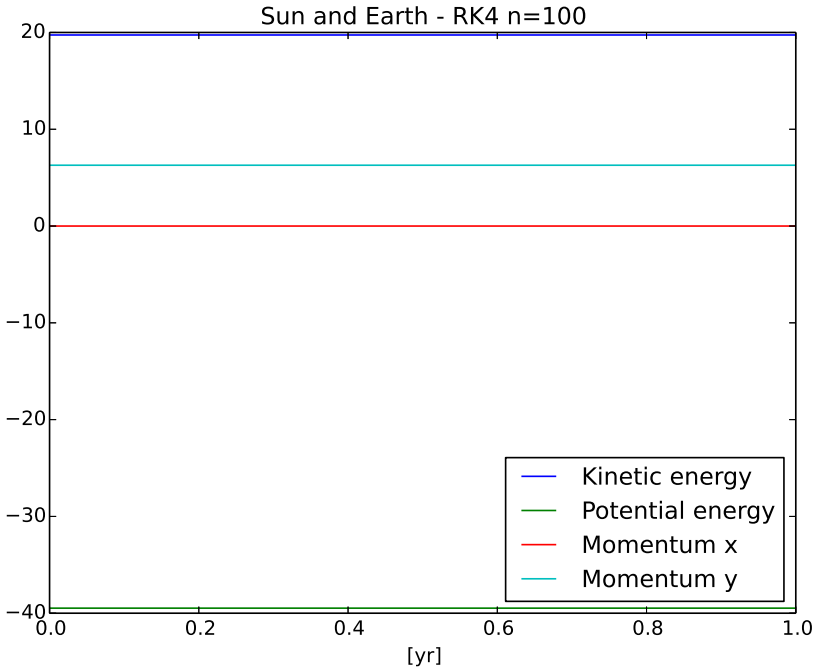

Figure 4.7: *This plot shows the conservation of energy and momentum for the earth and sun system with a RK4 solver. The initial velocity for the earth is assumed from the centripetal acceleration, and the sun's initial velocity is zero.*

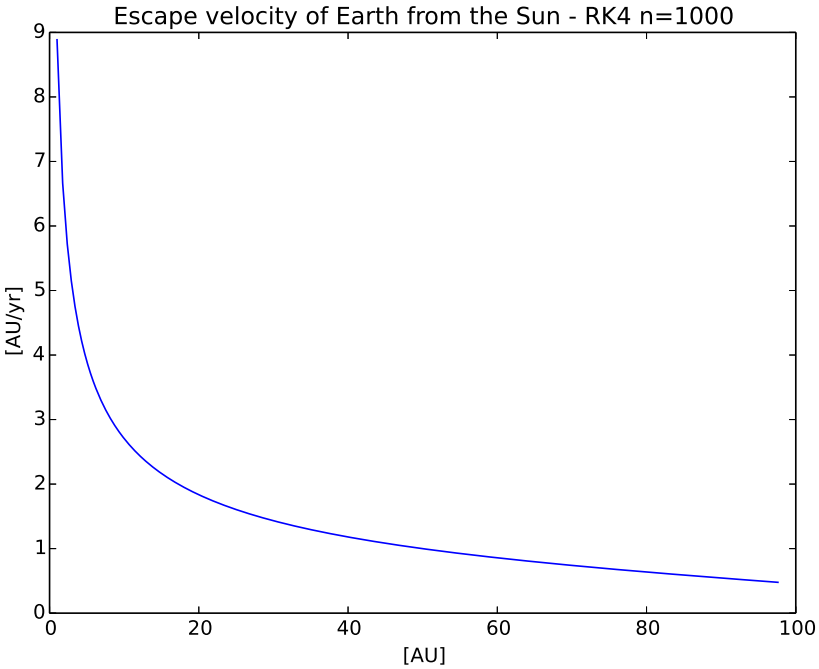

Figure 4.8: *The initial radial velocity calculated from* (2)*, where M is the mass of the sun and* r_0 *is the distance between the earth and sun.*

n	Verlet	RK4
10	3e-06	5e-06
20	4e-06	1.2e-05
50	9e-06	5.5e-05
100	2.3e-05	7.8e-05
1000	0.000232	0.000777
100000000	9.25578	35.6368

Table 4.1: *Computation time for the earth and sun system. RK4 is about 4 times slower than the Verlet solver with the same number of steps n. This comes from the fact that RK4 calculates the forces 4 times for each time step, where as Verlet calculates only one time.*

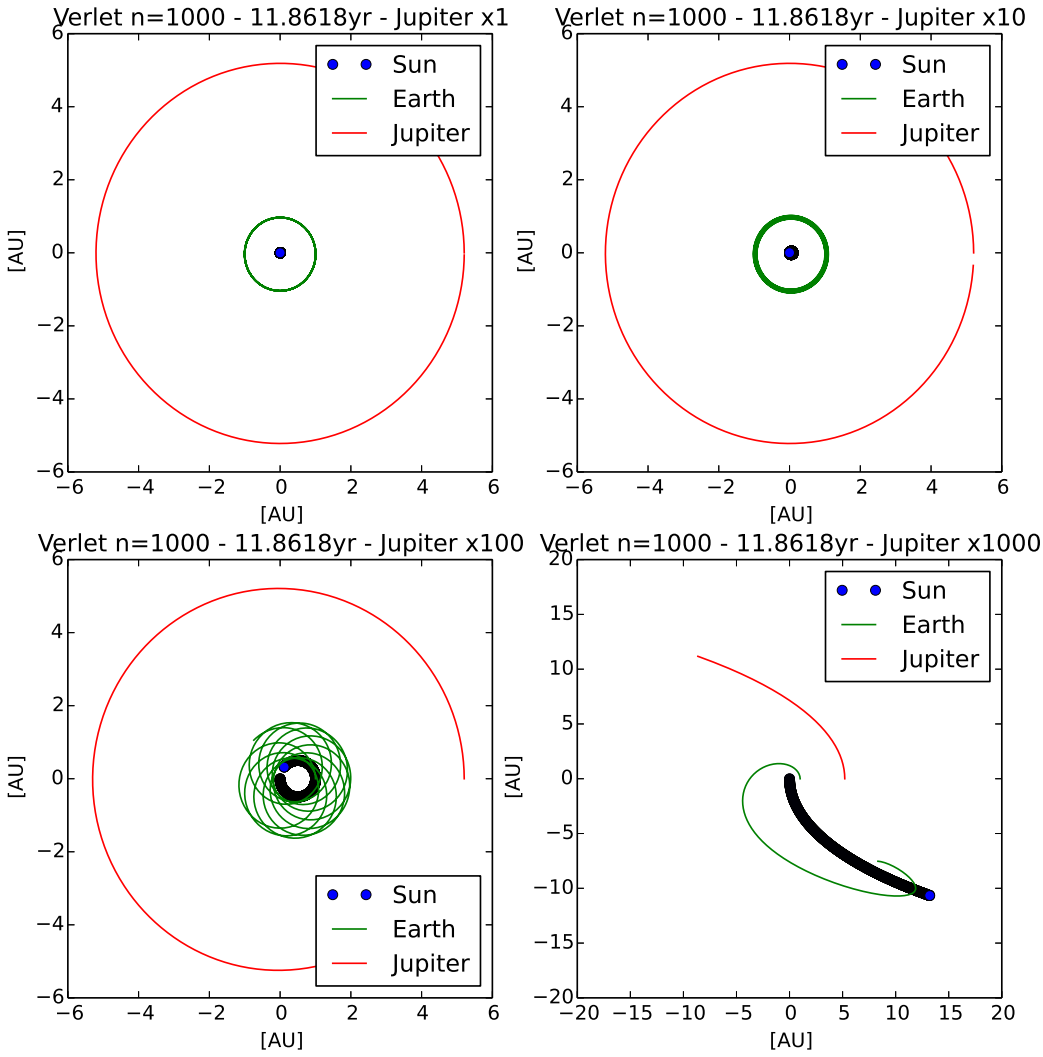

Figure 4.9: *Simulation of the orbit for the sun, earth and Jupiter system only with different masses of Jupiter, where the Verlet solver is used. The initial velocity for the earth and Jupiter is assumed from the centripetal acceleration, and the sun's initial velocity is such that the systems initial momentum is zero.*

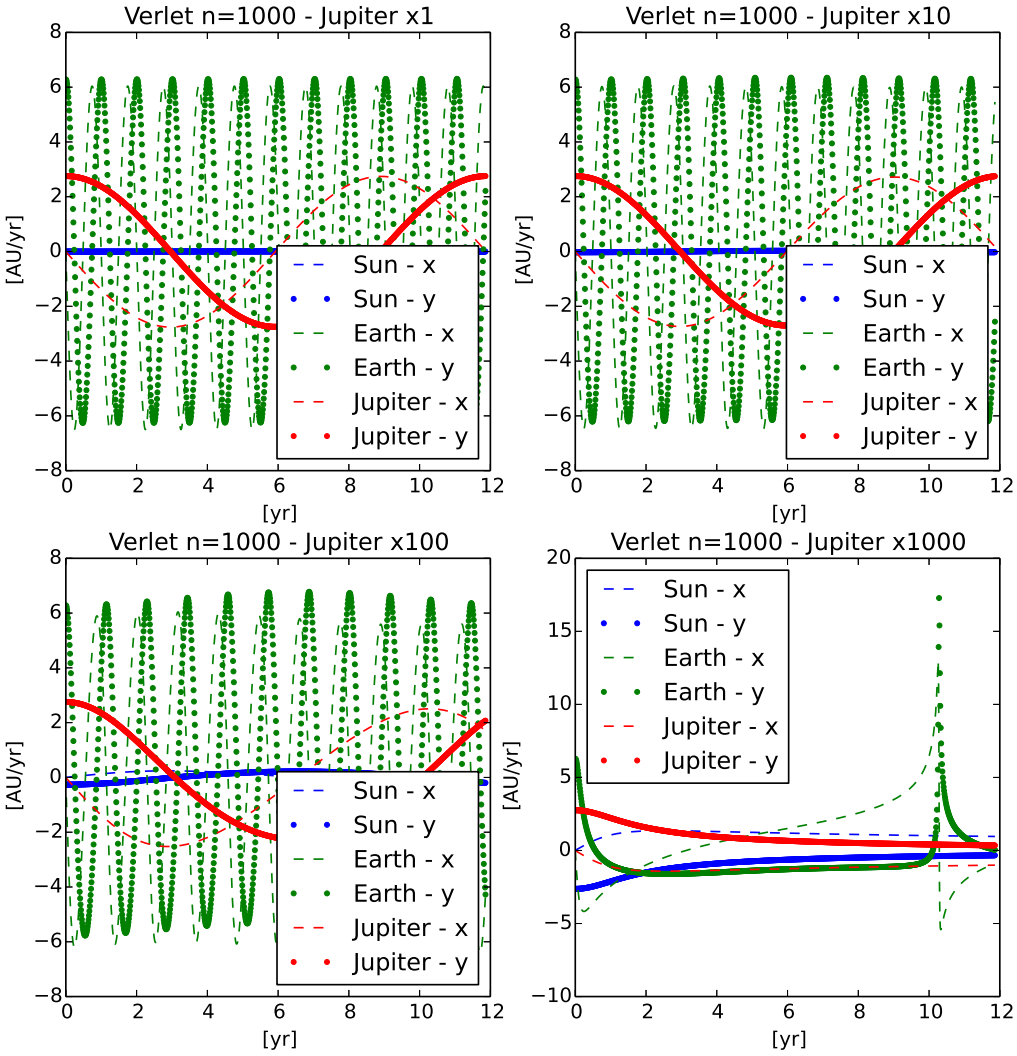

Figure 4.10: *Simulation of the periodicity for the sun, earth and Jupiter system only with different masses of Jupiter, where the Verlet solver is used. The initial velocity for the earth and Jupiter is assumed from the centripetal acceleration, and the sun's initial velocity is such that the systems initial momentum is zero.*

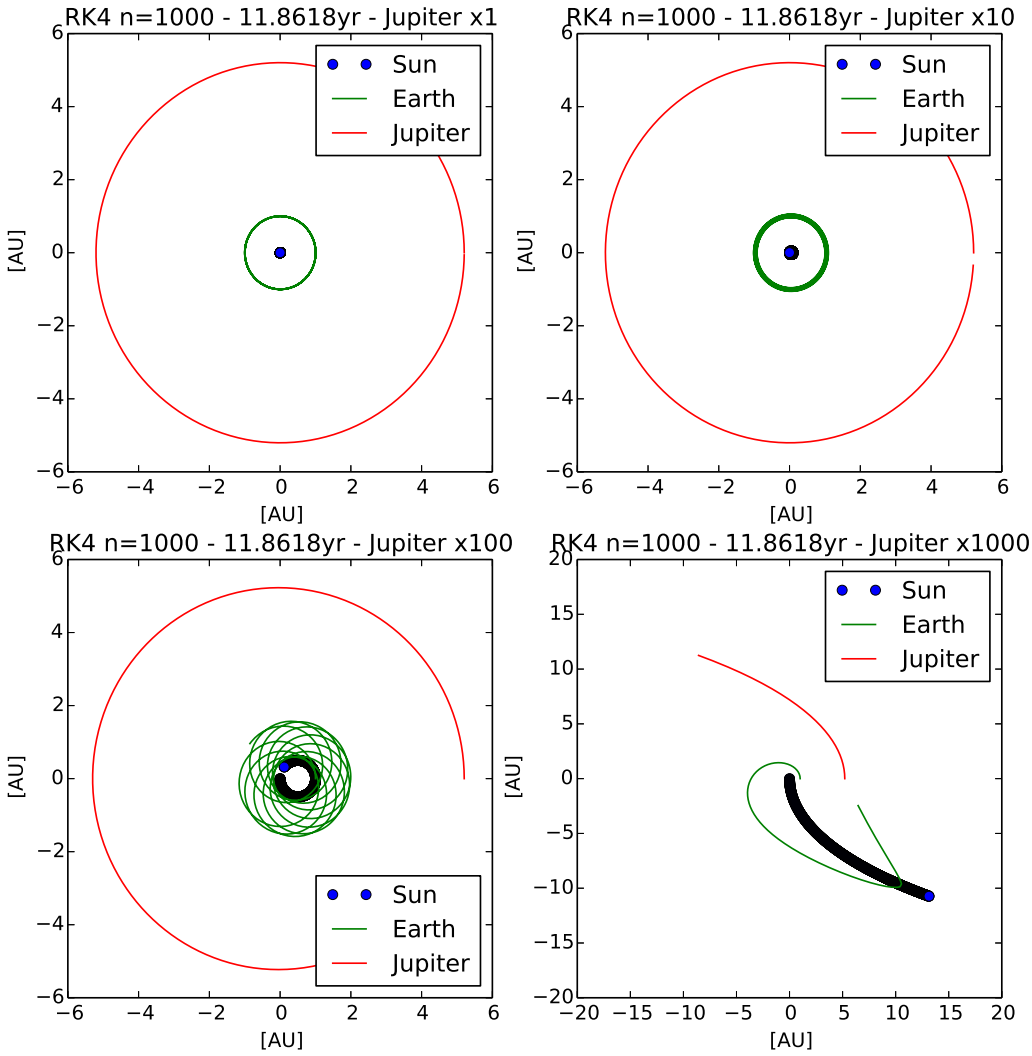

Figure 4.11: *Simulation of the orbit for the sun, earth and Jupiter system only with different masses of Jupiter, where the RK4 solver is used. The initial velocity for the earth and Jupiter is assumed from the centripetal acceleration, and the sun's initial velocity is such that the systems initial momentum is zero.*

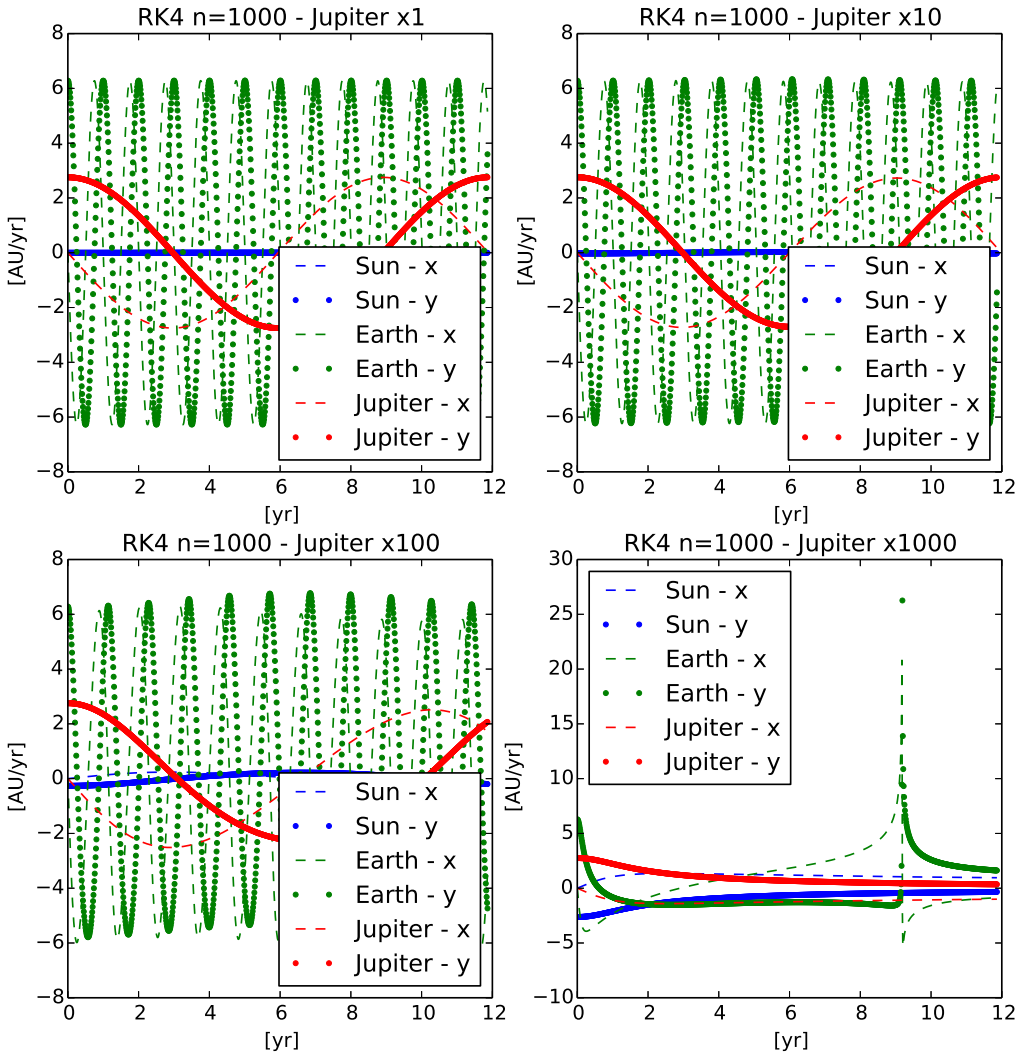

Figure 4.12: *Simulation of the periodicity for the sun, earth and Jupiter system only with different masses of Jupiter, where the RK4 solver is used. The initial velocity for the earth and Jupiter is assumed from the centripetal acceleration, and the sun's initial velocity is such that the systems initial momentum is zero.*

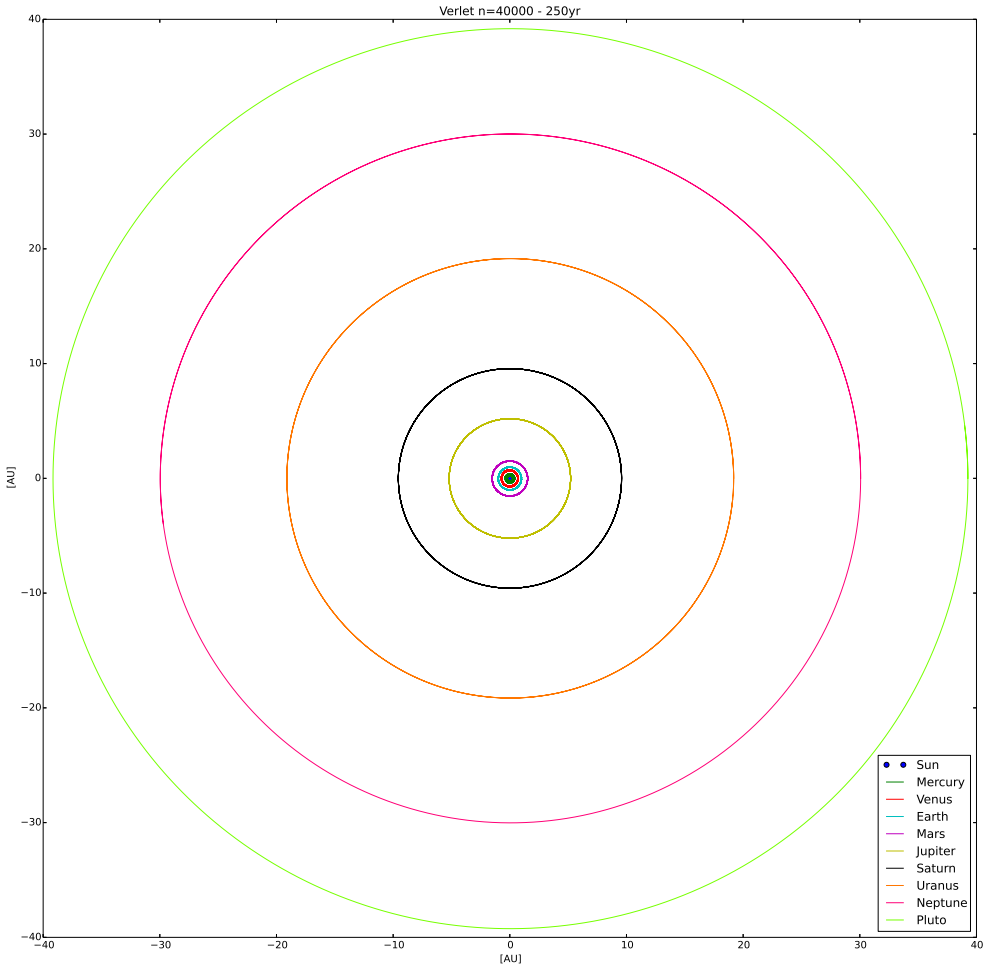

Figure 4.13: *Simulation of the orbits for the solar system, where the Verlet solver is used. The initial velocity for the plantes is assumed from the centripetal acceleration, and the sun's initial velocity is such that the solar systems initial momentum is zero.*

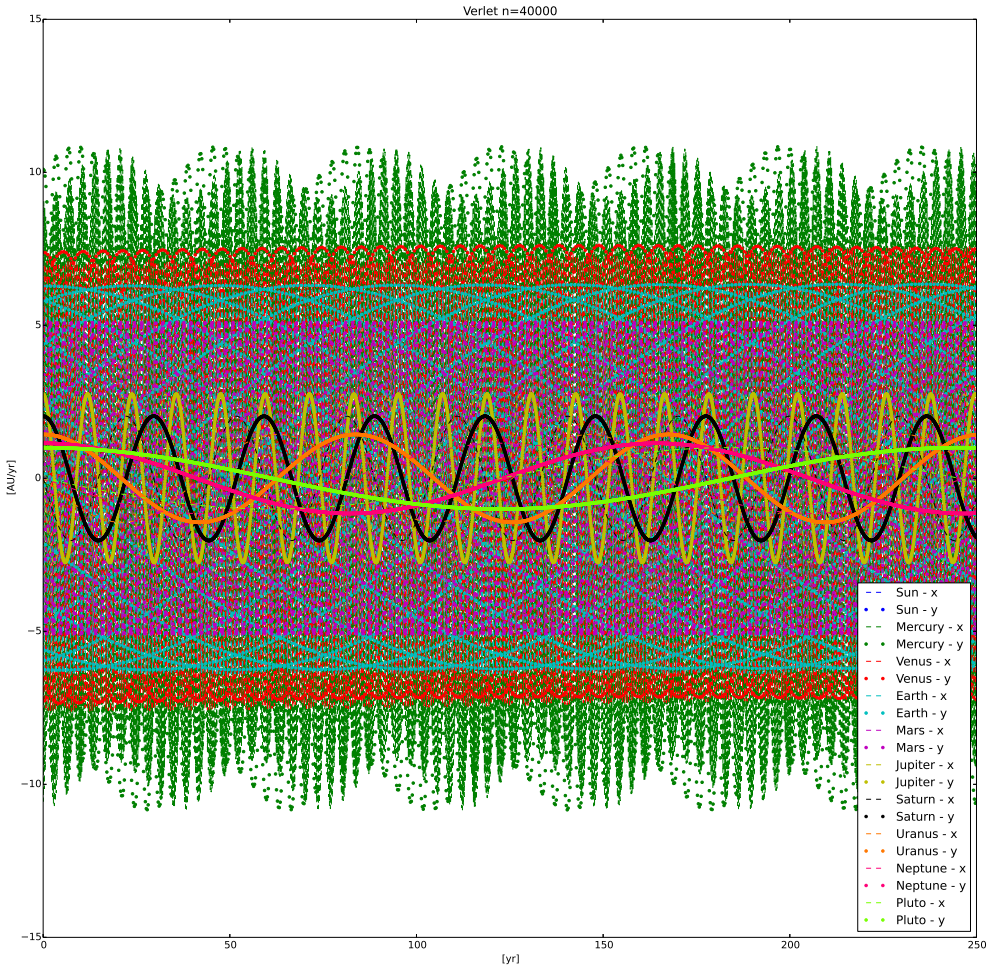

Figure 4.14: *Simulation of the periodicity for the solar system, where the Verlet solver is used. The initial velocity for the plantes is assumed from the centripetal acceleration, and the sun's initial velocity is such that the solar systems initial momentum is zero.*

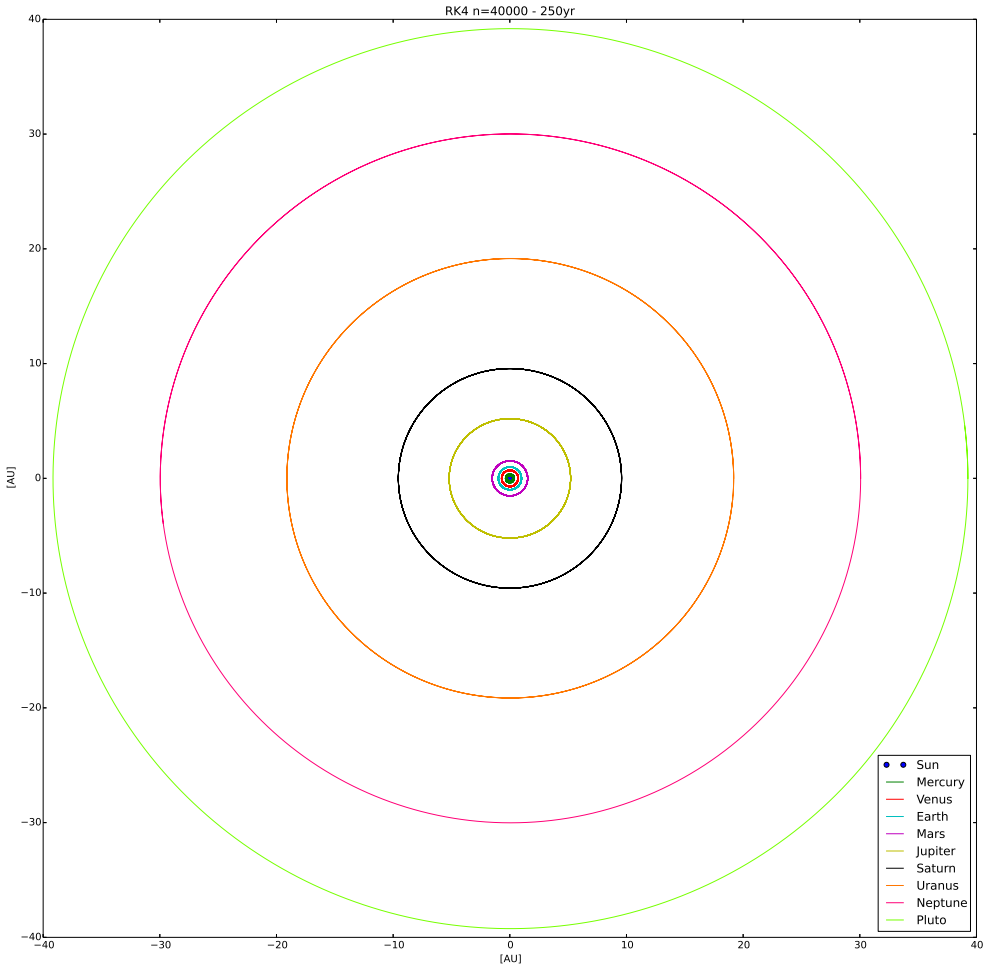

Figure 4.15: *Simulation of the orbits for the solar system, where the RK4 solver is used. The initial velocity for the plantes is assumed from the centripetal acceleration, and the sun's initial velocity is such that the solar systems initial momentum is zero.*

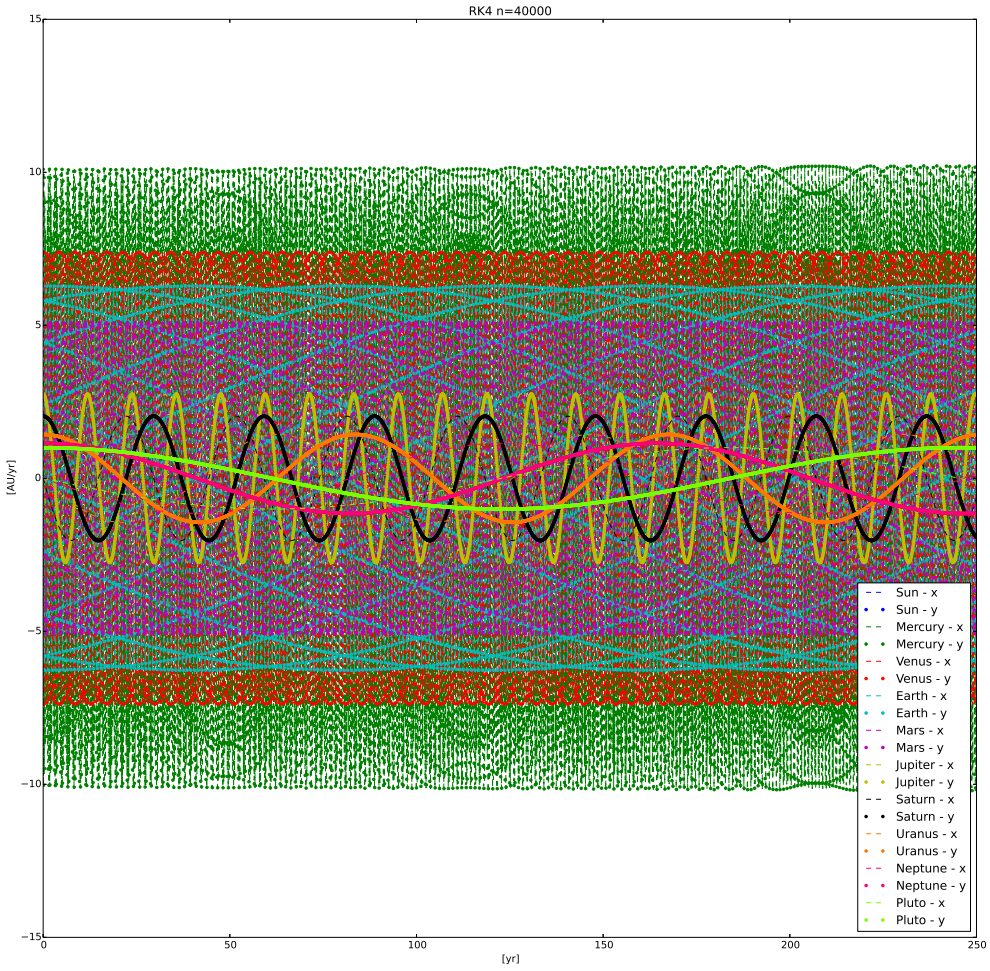

Figure 4.16: *Simulation of the periodicity for the solar system, where the RK4 solver is used. The initial velocity for the plantes is assumed from the centripetal acceleration, and the sun's initial velocity is such that the solar systems initial momentum is zero.*

n	Verlet	RK4
40000	0.075148	0.294784
400000	0.751995	2.96264
4000000	7.37535	29.4902

Table 4.2: *Computation time for the solar system. RK4 is about 4 times slower than the Verlet solver with the same number of steps n. This comes from the fact that RK4 calculates the forces 4 times for each time step, where as Verlet calculates only one time.*

5 Conclusion

The results for the solution of the solar system does not give an accurate representation of the actual solar system. The plants have a more elliptic orbit, and this is clearly evident from the orbit of Pluto. This is due to poor initial condition of the velocity. I have only used the centripetal acceleration as assumption of to find the initial velocity, which would result in a perfect circular motion in a two-body system. However the aim for this project is to study Verlet and RK4 method for solving second order ODE's.

We see from Figure 4.1 and Figure 4.3 that RK4 converges faster to the correct solution than the Verlet method. This comes from the fact that Verlet method has a local truncation error of order 4, where as the RK4 is of order 5. However the RK4 has an unpleasant side affect when we use it in a gravitational problem, and that is that the error tends to make the body spiral inwards. This we can see a tendency of in the first plot in Figure 4.3, where the earth's rotation is counter-clockwise. For the simulation for the whole solar system it showed that for to large time step with RK4, Mercury was thrown out of the solar system after few orbits around the sun. This was due to the decreasing orbit Mercury, which accelerated it to reach the escape velocity. The Verlet solver did not have this problem. We can see the reason for this in Figure 4.1, where the error tends to spiral outwards, however slowly and no dramatic changes happens like it did for the RK4 solver. For this reason the stability to solve a gravitational problem is better with the Verlet solver than the RK4 solver. And for the solar system a better approximation was reached with fewer time steps with the Verlet method than with the RK4, even though the RK4 has higher order of accuracy.

Another advantage for the Verlet method is the speed. As seen from Table 4.1 and Table 4.2 the Verlet method is about 4 times faster than RK4 method with the same number of time steps. And the time scalability of the Verlet method is also better, where we see from Table 4.1 and Table 4.2, when we increased with 8 bodies in the system, the computation time only increased 8 times with the same number of time steps. Whereas for the RK4 method the scalability was not as good, where the computation time increased 9 times when we added 8 bodies.

6 Attachments

The files produced in working with this project can be found at
https://github.com/Eimund/UiO/tree/master/FYS4150/Project%203/project3.

The source files developed are

1. Differential.h
2. Gravitation.h
3. project3.cpp

7 Resources

1. QT Creator 5.3.1 with C11
2. Eclipse Standard/SDK - Version: Luna Release (4.4.0) with PyDev for Python
3. Ubuntu 14.04.1 LTS

4. ThinkPad W540 P/N: 20BG0042MN with 32 GB RAM

References

[1] Morten Hjorth-Jensen, *FYS4150 - Project 3 - Building a model for the solar system*, University of Oslo, 2014

[2] Morten Hjorth-Jensen, *Computational Physics - Lecture Notes Fall 2014*, University of Oslo, 2014

[3] http://en.wikipedia.org/wiki/Newton%27s_law_of_universal_gravitation

[4] http://en.wikipedia.org/wiki/Gravitational_constant

[5] http://en.wikipedia.org/wiki/Astronomical_unit

[6] http://en.wikipedia.org/wiki/Year

[7] http://en.wikipedia.org/wiki/Newton%27s_laws_of_motion#Newton.27s_second_law

[8] http://en.wikipedia.org/wiki/Centripetal_force

[9] http://en.wikipedia.org/wiki/Escape_velocity

[10] http://en.wikipedia.org/wiki/Taylor_series

[11] http://en.wikipedia.org/wiki/Lagrange_polynomial

[12] http://en.wikipedia.org/wiki/Polynomial_interpolation

[13] http://en.wikipedia.org/wiki/Simpson%27s_rule

[14] http://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods

[15] http://en.wikipedia.org/wiki/Verlet_integration

[16] http://en.wikipedia.org/wiki/Sun

[17] http://en.wikipedia.org/wiki/Mercury_%28planet%29

[18] http://en.wikipedia.org/wiki/Venus

[19] http://en.wikipedia.org/wiki/Earth

[20] http://en.wikipedia.org/wiki/Mars

[21] http://en.wikipedia.org/wiki/Jupiter

[22] http://en.wikipedia.org/wiki/Saturn

[23] http://en.wikipedia.org/wiki/Uranus

[24] http://en.wikipedia.org/wiki/Neptune

[25] http://en.wikipedia.org/wiki/Pluto