11.3 N-body integrations

11.3.1 Newton's law

We'll discuss today an important real world application of numerical ODE integration: celestial mechanics. Celestial mechanics is the branch of astronomy that deals with the motions of any kind of celestial object. These motions are governed by Newton's law of gravitation. In some cases there are small correction due to general relativity, tides or radiation forces, which we will not include in our model.

Newtonian gravity says that a bodies of mass m_i feels another body of mass m_j, separated by the vector $\vec{r}$, with the following force

$$\vec{F}_{ij} = -\frac{G\, m_i m_j}{|\vec{r}_{ij}|} \cdot \frac{\vec{r}_{ij}}{|\vec{r}_{ij}|}$$

where G is the gravitational constant. The last term is simply telling us in which direction the force is directed (towards the other body). If there are N bodies in total, then the force is a sum over all bodies (except the body on which the force is exerted):

$$\vec{F}_i = -\sum_{\substack{j=0\\ i\neq j}}^{N-1} \frac{G\, m_i m_j}{|\vec{r}_{ij}|} \cdot \frac{\vec{r}_{ij}}{|\vec{r}_{ij}|}$$

To get an acceleration from a force, we use the well known law

$$\vec{F} = m\vec{a}$$

so that we end up with the acceleration on the i-particle being

$$\vec{a}_i = -\sum_{\substack{j=0\\ i\neq j}}^{N-1} \frac{G\, m_j}{|\vec{r}_{ij}|} \cdot \frac{\vec{r}_{ij}}{|\vec{r}_{ij}|}$$

The acceleration is the second time derivate of the position, so we can also write the last equation as

$$\ddot{\vec{r}}_i = -\sum_{\substack{j=0\\ i\neq j}}^{N-1} \frac{G\, m_j}{|\vec{r}_{ij}|} \cdot \frac{\vec{r}_{ij}}{|\vec{r}_{ij}|}$$

We have a total of N particles, thus we end up with a set of differential equation of second order

$$\ddot{\vec{r}}_i = F(\vec{r})$$

one for each $i = 0...N-1$ where $F(\vec{r})$ means that F depends on the r of all particles. All those equations are coupled. We already discussed how to solve this equation using a trick to convert it so first order.

$$\begin{pmatrix} r \\ \dot{r} \end{pmatrix}' = \begin{pmatrix} \dot{r} \\ F(r) \end{pmatrix}$$

Thus, we can simply use the Euler or midpoint method which we already know. Let's try that.

11.3.2 Initial conditions

As a test case, we use data from NASA's Horizon system to simulate the Solar system. Besides the major planets, we also include the comet 67P/Churyumov-Gerasimenko which is currently being visited by the Rosetta spacecraft. The data is given in the file `init.txt` and has one line per body and the format of name, followed by the x, y and z position and the x, y and z velocity. The units are km for the position, km/s for the velocity and the mass is given in kg.

```
sun 1.988544E+30 .491403347836458E+05
   -3.632715592552171E+05 -1.049148558556447E+04
   6.242516505718979E-03  1.163260794114081E-02
   -2.475374674040771E-04
mercury 3.302E+23 5.419423385170247E+07
   -1.625487416441774E+07 -6.231968220825911E+06
   4.381057475129131E+00  4.891120140478765E+01
   3.593097527852676E+00
venus 4.8685E+24 -2.256798710024889E+07
   1.046561273591759E+08  2.760223024328955E+06
   -3.431401385797417E+01 -7.710006447409605E+00
    1.875067390915588E+00
earth 5.97219E+24 -1.418287679667581E+08
   4.126884716814923E+07 -1.125285256157373E+04
   -8.830883522656004E+00 -2.868395352996171E+01
   -1.085239827735510E-05
mars 6.4185E+23 2.408279029085545E+07
   2.311290271817935E+08  4.261631465386531E+06
   -2.317767379354534E+01  4.521312752383141E+00
    6.638119814842468E-01
jupiter 1.89813E+27 -7.778414202838075E+08
   2.244518699271340E+08  1.647441732604697E+07
   -3.784015648894145E+00 -1.193534649902395E+01
    1.342303598962500E-01
saturn 5.68319E+26 -2.818805400917871E+08
   1.321479657136385E+09 -1.177469869993168E+07
   -9.963111372343775E+00 -2.038111913779875E+00
    4.325059188729626E-01
uranus 8.68103E+25 2.668275057715974E+09
   -1.368207197729832E+09 -3.965456607834578E+07
   3.057369253435637E+00  5.742539618944625E+00
   -1.841246675277081E-02
neptune 1.0241E+26 3.068964883542690E+09
   -3.290508769241064E+09 -2.965492893564129E+06
   3.938602703572163E+00  3.739029327757723E+00
   -1.671116647134993E-01
67P 0. -5.829716977332731E+08 -3.371344262751943E
   +08  3.008099774720960E+07 -1.302925387423112E
   +00 -1.173853622210837E+01 -7.989778520791224E
   -01
```

Code 50: The file `init.txt`

The plot below show the position of the planet on March 4th 2004.

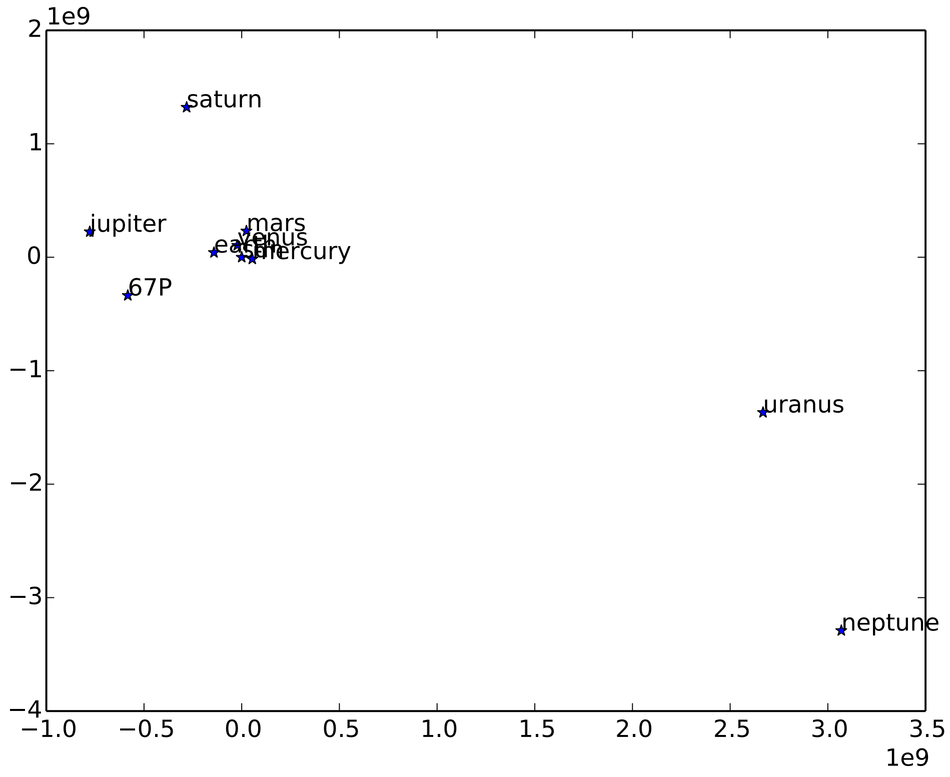

Figure 22: Positions of the Solar system planets on March 4th, 2014. The axes have units of km.

```
import math
names = []
m = []
x = []
y = []
z = []
with open("init.txt") as f:
    lines = f.readlines()
    for line in lines:
        rows = line.split()
        names.append(rows[0])
        m.append(float(rows[1]))
        x.append(float(rows[2]))
        y.append(float(rows[3]))
        z.append(float(rows[4]))

N = len(m)

import matplotlib
matplotlib.use('pdf')
import matplotlib.pyplot as plt
plt.plot(x,y,"*")
for i in xrange(N):
    plt.annotate(names[i],xy=(x[i],y[i]))
plt.savefig("plot_init.pdf")
```

Code 51: Python code producing the above plot

11.3.3 Integrating the Solar system with the Euler Method

Let's first integrate the system using the Euler method. Remember, this is a first order method and not very accurate. The planets go around the Sun on a timescale of years, so let's use a timestep significantly smaller, say one day.

The code that implements the Euler method is below.

```
import math
m = []
x = []
y = []
z = []
vx = []
vy = []
vz = []
with open("init.txt") as f:
    lines = f.readlines()
    for line in lines:
        rows = line.split()
        m.append(float(rows[1]))
        x.append(float(rows[2]))
        y.append(float(rows[3]))
        z.append(float(rows[4]))
        vx.append(float(rows[5]))
        vy.append(float(rows[6]))
        vz.append(float(rows[7]))

G = 6.67384e-20
N = len(m)
t = 0.
t_max = 1.*365.*24.*60.*60.
dt = 1.*24.*60.*60.
while t<t_max:
    xo = x[:]
    yo = y[:]
    zo = z[:]
    for i in xrange(N):
        x[i] += vx[i] * dt
        y[i] += vy[i] * dt
        z[i] += vz[i] * dt
    for i in xrange(N):
        ax = 0.
        ay = 0.
        az = 0.
        for j in xrange(N):
            if i!=j:
                dx = xo[i]-xo[j]
                dy = yo[i]-yo[j]
                dz = zo[i]-zo[j]
                r = math.sqrt(dx*dx + dy*dy + dz*dz)
                a = -G*m[j]/(r*r)
                ax += a/r * dx
                ay += a/r * dy
                az += a/r * dz
        vx[i] += ax * dt
        vy[i] += ay * dt
        vz[i] += az * dt
    t = t + dt
    for i in xrange(N):
        print x[i],y[i],z[i]
```

Code 52: Python code of the Euler method, integrating the Solar system.

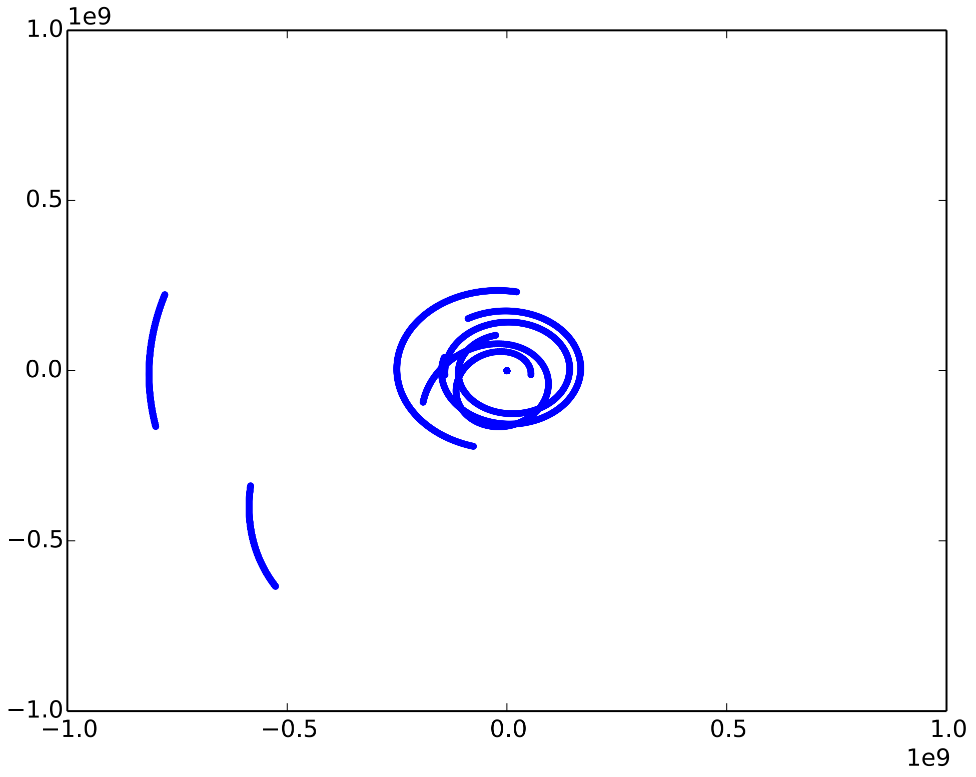

Figure 23: Orbits of the Solar system planets.

```
import math
x = []
y = []
z = []
with open("pos.txt") as f:
    lines = f.readlines()
    for line in lines:
        rows = line.split()
        x.append(float(rows[0]))
        y.append(float(rows[1]))
        z.append(float(rows[2]))

import matplotlib
matplotlib.use('pdf')
import matplotlib.pyplot as plt
plt.plot(x,y,".")
plt.xlim((-1e9,1e9))
plt.ylim((-1e9,1e9))
plt.savefig("plot_orbits.pdf")
```

Code 53: Python code plotting the orbits of the Solar system bodies This produces the above plot.

This is a reasonable result for a first order method. But not adequate for any real application. If you look closely at the orbit of Mercury, you can see that it spirals in. If you can see this error by eye, it will not take make orbits for Mercury to spiral into the sun.

So what can we do? Well, we could try the midpoint method. This would give us a better result, but we would still get planets spiralling into the star. So let's me introduce you to another method, very often used for this kind of second order equation: Leap-Frog.

11.3.4 Integrating the Solar system with Leap Frog

The basic idea of the leapfrog method is to stagger the updates of positions and velocity in time. This makes the method symmetric. We already encountered that this sort of symmetry makes the method more accurate. Thus, it is not surprising that leapfrog is a second order method. One timestep in the leapfrog method is as follows

$$\begin{aligned} x_{n+1/2} &= x_n + \frac{1}{2} dt\, v_n \\ v_{n+1} &= v_n + dt\, F(x_{n+1/2}) \\ x_{n+1} &= x_{n+1/2} + \frac{1}{2} dt\, v_{n+1} \end{aligned}$$

So we first update the positions for half a timestep, then the velocities for one full timestep and finally the positions for another half a timestep. Remember, v is just the first derivate of x, so rewriting this in terms of one variable x and it's derivative $\dot{x}$ we get

$$\begin{aligned} x_{n+1/2} &= x_n + \frac{1}{2} dt\, \dot{x}_n \\ \dot{x}_{n+1} &= \dot{x}_n + dt\, F(x_{n+1/2}) \\ x_{n+1} &= x_{n+1/2} + \frac{1}{2} dt\, \dot{x}_{n+1} \end{aligned}$$

Modifying our Euler Method to do the Leap-Frog method is surprisingly easy and is listed below.

```
import math
m = []
x = []
y = []
z = []
vx = []
vy = []
vz = []
with open("init.txt") as f:
    lines = f.readlines()
    for line in lines:
        rows = line.split()
        m.append(float(rows[1]))
        x.append(float(rows[2]))
        y.append(float(rows[3]))
        z.append(float(rows[4]))
        vx.append(float(rows[5]))
        vy.append(float(rows[6]))
        vz.append(float(rows[7]))

G = 6.67384e-20
N = len(m)
t = 0.
t_max = 1.*365.*24.*60.*60.
dt = 1.*24.*60.*60.
while t<t_max:
    for i in xrange(N):
        x[i] += vx[i] * dt/2.
        y[i] += vy[i] * dt/2.
        z[i] += vz[i] * dt/2.
    for i in xrange(N):
        ax = 0.
        ay = 0.
        az = 0.
        for j in xrange(N):
            if i!=j:
                dx = x[i]-x[j]
                dy = y[i]-y[j]
                dz = z[i]-z[j]
                r = math.sqrt(dx*dx + dy*dy + dz*dz)
                a = -G*m[j]/(r*r)
                ax += a/r * dx
                ay += a/r * dy
                az += a/r * dz
        vx[i] += ax * dt
        vy[i] += ay * dt
        vz[i] += az * dt
    for i in xrange(N):
        x[i] += vx[i] * dt/2.
        y[i] += vy[i] * dt/2.
        z[i] += vz[i] * dt/2.
    t = t + dt
    for i in xrange(N):
        print x[i],y[i],z[i]
```

Code 54: Python code of the Leap-Frog Method.

The orbits are shown in the following plot.

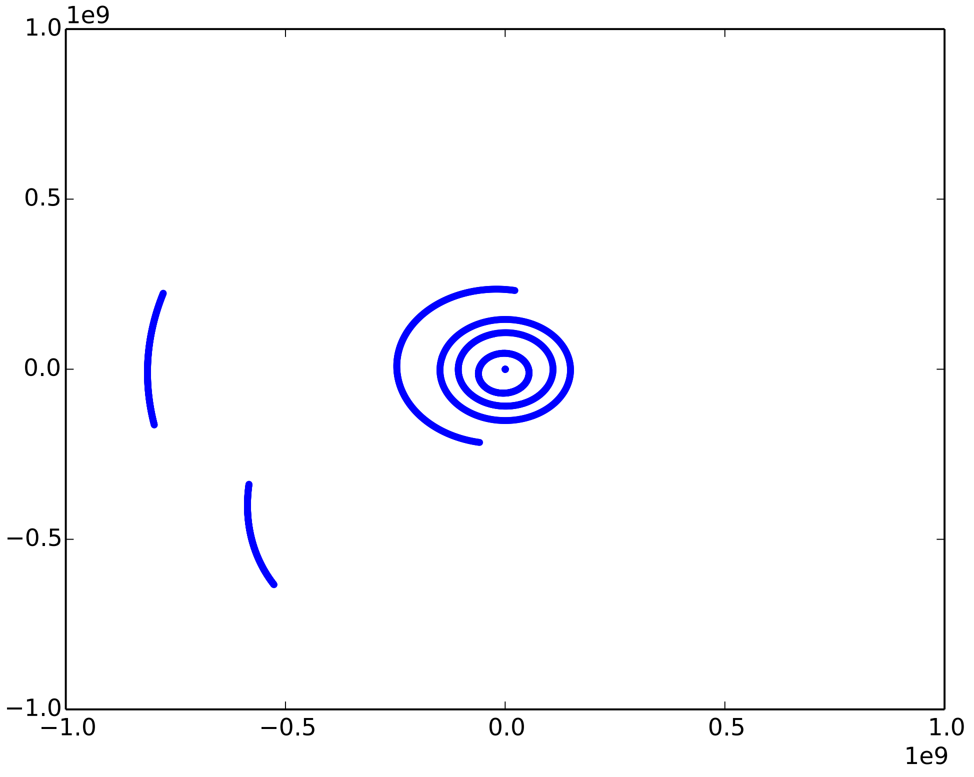

Figure 24: Orbits of the Solar system planets integrated with leapfrog and $dt = 1$day.

There are a few more advantages to the leapfrog method. First, there is only one force function evaluation per timestep. This makes it as fast as the Euler method, but much more accurate! Second, leapfrog is not only second order and time symmetric, but also symplectic. This is a property that we cannot fully discuss in this course. However, keep the name in mind. It is an incredibly powerful class of geometric numerical integrators which allow you to integrate Hamiltonian systems very efficiently.

End of lecture 10.