

Lab 3 - Modeling the Solar System (part I)

Thursday 14 September, 2006 - Due: Thursday 21 September

The aim of this lab is to **get you started** with the construction of a solar system, using some of the ideas that you have already been introduced to. In particular... **V/Python**: 3D-graphics, 2D-graphing, lists, and modules;
Numerical methods: integrators: Euler and improvements on Euler;
Physics: many-particle kinematics and dynamics.

But don't worry... we will develop our Solar System slowly... probably continuing on into the next lab.

As always, please include in your writeup any sections of Python code you write.

1 a short pause

Take about 30 minutes to finish up the lab you started last week.

2 a short time dt with Euler

Below is the program shown in Lecture 3 which demonstrates the Euler integration method:

$$\begin{aligned}x_{n+1} &= x_n + v_n dt \\v_{n+1} &= v_n + a_n dt\end{aligned}$$

We want to study the behavior of this algorithm for various choices of dt .

Copy-paste or type this program (which spills over onto the next page) into an IDLE window, and save this as **energy.py**. Verify the indentation!

```
from visual import *
from visual.graph import *

scene=display(x=0,y=0,width=400,height=400,title="simulation")
scene.autoscale=0
scene.range=10.0

pic=gdisplay(x=400,y=0,width=400,height=400,title="x vs t",
            xtitle="t",ytitle="x",xmax=20.0,xmin=0,ymin=0.0,ymax=12.0,
            background=color.white, foreground=color.black)
xplot=gcurve(color=color.blue)

def force(pos,vel,t):
    result = vector(0,0,0)
    result.x=-1.0*pos.x
    result.y=0.0
```

```

    result.z=0.0
    return result

ball=sphere(radius=0.5,pos=vector(4.0,0.0,0),
            track=curve(radius=0.1,display=scene),mass=1.0,display=scene)
ball.vel=vector(0,0,0)

dt=0.1
t=0

while (t<20.0) :
    rate(100)
    t=t+dt
#euler alg.
    ball.pos=ball.pos+ball.vel*dt
    ball.vel=ball.vel+force(ball.pos,ball.vel,t)/ball.mass*dt
    ball.track.append(pos=ball.pos)
    energy=0.5*ball.mass*ball.vel.x*ball.vel.x+0.5*ball.pos.x*ball.pos.x
    xplot.plot(pos=(t,energy))

```

Recall that this program models a simple harmonic oscillator, with force law: $\vec{F} = -k\vec{x}$, and graphs the total energy $E = \frac{1}{2}mv^2 + \frac{1}{2}kx^2$ as a function of time. (The program uses a spring constant $k = 1.0$.)

(Q1)★ Using the parameters defined in the program, calculate the initial total energy E_0 .

Although *physically* the simple harmonic oscillator has a constant total energy, our *computational* model using the Euler integration method has an oscillating total energy.

(Q2)★ Run this program with $dt = 0.1$ and “read off the graph” the valley-to-peak amplitude ΔE of the oscillation in the total energy. (Enlarge the graph window, if necessary.)

dt	$(\Delta E)/E$
0.4	
0.2	
0.1	
0.05	
0.02	
0.01	

Complete this table for $dt \geq 0.05$:

With $dt \leq 0.05$, it is more difficult to determine ΔE by reading it off the graph.

(Q3 at home)★ Insert a line of code `print energy` before the `xplot.plot()` line. Repeat the $dt = 0.05$ case, and use the values in the shell window to determine ΔE . Complete the rest of the table.

(Q4 at home)★ Use the following example program (substituting your own table values) to make a plot of your table above.

```

from visual import *
from visual.graph import *

gdisplay(xtitle='dt', ytitle='dE/E',
         background=color.white, foreground=color.black )

points = [(1,2), (3,4), (-4,2), (-5,-3)]

data = gdots(pos=points, color=color.blue)

```

Include a screen-capture of the plot of your table in your report.

3 using a list in the Euler program

The tedium of playing with variations of a parameter can be simplified using a **list** and a **for**-loop.

(Q5)★ Make the following modifications to the **energy.py** program. Replace all of the lines after **ball=sphere(...)** with these lines:

```

for dt in [0.01,0.02,0.05,0.1,0.2,0.4]:
    ball.vel=vector(0,0,0)
    ball.pos=vector(4.0,0.0,0)
    #xplot=gcurve(color=color.blue)

    t=0
    while (t<20.0) :
        rate(100)
        t=t+dt
        #euler alg.
        ball.pos=ball.pos+ball.vel*dt
        ball.vel=ball.vel+force(ball.pos,ball.vel,t)/ball.mass*dt
        ball.track.append(pos=ball.pos)
        energy=0.5*ball.mass*ball.vel.x*ball.vel.x+0.5*ball.pos.x*ball.pos.x
        xplot.plot(pos=(t,energy))

```

Read this explanation before you run the program.

- We have replaced the **dt=0.1** statement (which you modified several times to study the effect of *dt*) by a **for**-loop running over a list of the requested *dt* values.
- *Within* this newly constructed loop, we wish to essentially execute the rest of the original program. So everything [in particular, the **while**-loop] is now indented to be executed during each iteration of the loop. (*By the way, a great way to indent a block of code is to highlight the block, then hit the [TAB]-key.*)
- Since we want to restore some key variables (like **ball.vel**, **ball.pos**, **t**) to their initial conditions, we include them in the loop. [Admittedly, the code has some redundancies... but this is easiest modification.]

- Finally, since the `t` variable is reset to 0 during each iteration, the graph of the total energy would jump back to $t = 0$, causing some unwanted streaks in the display. To see this, **run the program now**.

To avoid these streaks, remove the comment symbol `#` preceding the `xplot=gcurve(...)` statement. This statement creates a new `gcurve` and calls it `xplot`. The previous `gcurve` has now lost its name (or reference), and thus cannot easily be updated—which is fine because we are done with that `gcurve` from the previous value of `dt`.

4 Building your own Solar System

Finally, let's modify the code presented in Lecture 3 to model a “mini Solar System” consisting of the Sun and the Earth and, later, the Moon.

- **the `integrator` module:**

First, create a new file called `integrator.py`:

```
from visual import *

G=6.673e-11
## Force on a due to b
def force(a,b):
    diff=b.pos-a.pos
    return G*b.mass*a.mass*norm(diff)/diff.mag2

## Finds acceleration of a due to all objects b
def totalacc(a,objlist):
    sum_acc=vector(0,0,0)
    for b in objlist:
        if (a!=b):
            sum_acc=sum_acc+force(a,b)/a.mass
    return sum_acc

## Finds total acceleration on all objects
def update_acceleration(objlist):
    for i in objlist:
        i.acc=totalacc(i,objlist)

## updates positions and track of each object
def update_position(objlist, dt):
    for i in objlist:
        i.pos=i.pos+dt*i.velocity
        i.track.append(pos=i.pos)

## update velocity of each object
def update_velocity(objlist, dt):
    for i in objlist:
        i.velocity=i.velocity+dt*i.acc
```

(The only difference between this program and the one presented in the lecture is the use of a more realistic value of the Gravitational constant G .)

A **module** is a convenient way to make useful functions available to other Python programs, without having to copy-paste the code for the functions into each client program. In addition, it could be a good way to update the functions and not have to replicate the changes in each client program. For example, `visual` is the [fancy] module that gives a Python program 3D-visualization functions.

Our `integrator` module is a simple module to give the integration methods needed to calculate the net-forces, the accelerations, the velocities, and the positions of objects in our Solar System. The meaning of each function should be readable from the code. However, let me say a few words about the use of *lists* in this module.

- By using a *list*, these functions can be applied [in principle] to an arbitrary number of “bodies”. (In fact, that number need not be constant throughout the program. You might imagine a “planet” somehow exploding into several pieces, which can change the list of bodies.) While adding bodies will certainly increase the number of computations required, the logical structure of the program is unchanged.

- **the modified gravity code:**

Now, create a new file called `sun_earth.py`:

```
from visual import *
from integrator import *

AU=149.6e9 #earth-sun orbital-radius
MU=384.4e6 #moon-earth orbital-radius

sun_mass=2e30
sun_radius=6.96e8

earth_mass=6e24
earth_radius=6.37e6
earth_vel=2*math.pi*AU/(365.25*24.*60.*60.)

moon_mass=7.36e22
moon_radius=1.74e6
moon_vel=2*math.pi*MU/(27.32*24.*60.*60.)

scene.background=color.white
scene.autoscale=0
scene.range=2*AU

sun =sphere(pos=(0, 0,0), velocity=vector(0,0,0),
            mass=sun_mass, radius=0.1*AU, color=color.yellow)
earth=sphere(pos=(AU, 0,0), velocity=vector(0,earth_vel,0),
            mass=earth_mass, radius=0.001*AU, color=color.cyan)
```

```

# create a list of gravitating objects
bodies=[sun,earth]

for b in bodies:
    b.acc=vector(0,0,0)
    b.track=curve(color=b.color)

dt=30.*60.
while True:
    rate(100)
    update_position(bodies,dt)
    update_acceleration(bodies)
    update_velocity(bodies,dt)
    scene.center=earth.pos    #center the view on the earth

```

- First, note the statement `from integrator import *`. The functions in that `integrator` module are now accessible in your program.
- I looked up some physical constants and expressed them in SI-units. It was convenient to give them descriptive names that I used later when creating the `sphere` for each body.
- To be fully realistic, I should have set the radii of the spheres equal to the true radii of corresponding bodies. However, they would be too small to see on our display. So, we have cheated by giving the spheres artificial radii, which are conveniently described in units of `AU`.
- After building up the *list* of bodies, I added two attributes `acc` and `track`. Note how the color of the `curve` is matched with the color of the `sphere`. Although these attributes could have been included in the original `sphere(..)` statements, using a list for these attributes was more efficient.
- Since we are using realistic numbers, I had to choose a corresponding reasonable size for the simulation increment of time `dt`... 30 min, which could be executed at a `rate` of at most 100 times per second. So [on a capable computer], we could have simulated 3000 min \approx 2.08 days each second. This amounts to a full orbit of the earth in about 3 minutes .
- The line `scene.center=earth.pos` places the earth in the center of the view so that you can zoom-in and -out on the earth and its orbit.

(Q6)★ –Add the moon to your mini Solar System. Give the moon its true mass, an initial position of `AU+MU` along the \hat{x} -axis, an initial velocity of `earth_vel+moon_vel` along the \hat{y} -axis (tangent to the circular orbit), a reasonably-artificial display radius, and a red color. You might wish to zoom-in to see the earth and moon interact with each other, as well as with the sun.