

ORF 201  
Computer Methods in Problem Solving

Lab 4: Solar System Simulation

Due Sunday, March 6, 11:59 pm

---

1. INTRODUCTION

This assignment is about simulating the motion of heavenly bodies as they move under the influence of gravity. We will first describe the physics that underlies the simulation. If you are rusty on physics (or haven't taken the relevant course yet), have no fear since all the important formulas will be carefully laid out and you should be able to program them even if you don't fully understand them.

2. NEWTON'S LAW OF GRAVITATION

Newton's *law of gravitation* says that two masses,  $m_1$  and  $m_2$ , experience an attractive force whose magnitude is given by

$$F = G \frac{m_1 m_2}{r^2}.$$

Here  $G$  is a universal constant (called the *gravitational constant*, whose value is  $6.67 \times 10^{-11} \text{Nm}^2/\text{kg}^2$ ) and  $r$  denotes the distance between the two masses. Of course, force is really a vector—the above formula only specifies its length. The assertion that the force is *attractive* determines its direction. Indeed, the magnitude given above must be multiplied times a unit vector pointing in the correct direction. Let  $\mathbf{p}_1 = (x_1, y_1)$  be the position vector<sup>1</sup> for body 1 and let  $\mathbf{p}_2 = (x_2, y_2)$  denote the position vector for body 2 (these position vectors are relative to some arbitrary coordinate system). Then, the distance  $r$  between the two bodies is given by

$$r = \|\mathbf{p}_1 - \mathbf{p}_2\| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Consider the force,  $\mathbf{F}_1$ , on body 1 caused by the gravitational attraction of body 2. A unit vector pointing toward body 2 is given by

$$\frac{\mathbf{p}_2 - \mathbf{p}_1}{r}$$

---

<sup>1</sup>We use boldface to denote vectors. In this lab all vectors are assumed to be two-dimensional.

and so the force vector is given by

$$\begin{aligned}\mathbf{F}_1 &= G \frac{m_1 m_2}{r^2} \frac{\mathbf{p}_2 - \mathbf{p}_1}{r} \\ &= G \frac{m_1 m_2}{r^3} (\mathbf{p}_2 - \mathbf{p}_1)\end{aligned}$$

### 3. NEWTON'S SECOND LAW OF MOTION

In addition to his law of gravitation, Newton also formulated laws of motion. *Newton's second law of motion* says that the acceleration  $\mathbf{a} = (a^x, a^y)$  on a body is equal to the force  $\mathbf{F} = (F^x, F^y)$  divided by the mass:

$$\mathbf{a} = \mathbf{F}/m.$$

Therefore, the acceleration of body 1 due to the gravitational pull of body 2 is given by

$$(1) \quad \mathbf{a}_1 = \mathbf{F}_1/m_1 = G \frac{m_2}{r^3} (\mathbf{p}_2 - \mathbf{p}_1).$$

Similarly, the acceleration of body 2 due to the gravitational pull of body 1 is given by

$$(2) \quad \mathbf{a}_2 = G \frac{m_1}{r^3} (\mathbf{p}_1 - \mathbf{p}_2).$$

### 4. COMPUTER SIMULATION

To make a computer simulation, we discretize time and look at successive discrete points in time. We can think of this as a sequence of *snapshots* that are displayed in rapid succession to make a movie. We assume that the time interval between these snapshots is a **small** number  $\Delta t$ . We also assume that positions and accelerations are computed at the endpoints of the time intervals but that velocities are computed at the midpoints of the time intervals. Finally, we assume that we are given initial position and velocity vectors for both bodies and that the initial velocity vector corresponds to the midpoint of the first time interval.

To understand how the calculations go, suppose that the simulation has been running for awhile and let's see how one updates the position, velocity, and acceleration over one time interval. Let  $t$  denote the current time, which is at the beginning of one of these discrete time intervals. We have the position at time  $t$  and the velocity at time  $t + \Delta t/2$ . We assume that the velocity at this midpoint time is a good approximation to the average velocity over the entire time interval  $[t, t + \Delta t]$  and so update the position vectors by incrementing them by the velocity times the length of the time interval:

$$\begin{aligned}\mathbf{p}_1(t + \Delta t) &= \mathbf{p}_1(t) + \mathbf{v}_1(t + \Delta t/2)\Delta t \\ \mathbf{p}_2(t + \Delta t) &= \mathbf{p}_2(t) + \mathbf{v}_2(t + \Delta t/2)\Delta t.\end{aligned}$$

Now that new positions are known (i.e., at the beginning of the next time interval), we can use equations (1) and (2) to compute acceleration vectors at the new time (that is,  $t + \Delta t$ ). Given the acceleration vector  $a_1(t + \Delta t)$ , we assume that it is a good approximation to the

average acceleration over the midpoint-to-midpoint time interval  $[t + \Delta t/2, t + 3\Delta t/2]$  and update the velocity as follows:

$$\mathbf{v}_1(t + 3\Delta t/2) = \mathbf{v}_1(t + \Delta t/2) + \mathbf{a}_1(t + \Delta t)\Delta t.$$

The velocity of body 2 is updated in a similar fashion:

$$\mathbf{v}_2(t + 3\Delta t/2) = \mathbf{v}_2(t + \Delta t/2) + \mathbf{a}_2(t + \Delta t)\Delta t.$$

At this point, we iterate the process just described to carry out the simulation.

## 5. GETTING STARTED

First, you need to create a folder on your H: drive:

```
H:\public_html\JAVA\ORF201\solar
```

Then, use your favorite web browser to download the following zip file:

```
http://www.princeton.edu/~orf201/JAVA/ORF201/solar/lab4.zip
```

Save this file in the `solar` folder you created on your H: drive and then unzip the file by double clicking on it in Windows Explorer. The unzipper should create two new java files:

```
Solar.java, Body.java
```

and a number of image and text data files that will be explained later. All these files must be in your `solar` folder.

## 6. COMPILING

Before you actually begin changing and/or adding on to the code, check to see that you can compile and run the code as given. On Window Explorer find `H:/public_html/JAVA/ORF201/solar`, right-click on `Body.java` and load it onto TextPad. Function key `Control+1` will start the compiler. The code should compile without errors. Repeat the same steps for the file `Solar.java`. To run the application, having `Solar.java` shown in the main window of TextPad, hit `Control+2`. After a pause, a frame window should pop up with a starry image as the background. At the moment, nothing else happens. That is because the main parts of the code in `Body.java` and `Solar.java` have been stripped out. It is your job to fill them in according to the instructions given below.

## 7. PROGRAMMING NOTES:

In this assignment, you are asked to do four things: implement a `Body` object, read-in some data and create instances of that `Body` object, simulate the gravitational interaction among the bodies and draw the animation of the resulting movement of objects.

**7.1. Implement a `Body` class to contain information about a heavenly body.** You can do this by filling in the `Body.java` file provided. You will need to include in this class at least the following data members: the  $x$  and  $y$  coordinates of the body's position, the  $x$  and  $y$  components of its velocity, its mass and an image to be associated to the body.

**7.2. Instantiate an array of `Body` objects.** In order to instantiate an array of these objects, we will provide you with data stored in some files. The usage of these data files, along with the several images referred to in them, is a courtesy of Profs. Robert Sedgewick and Kevin Wayne, who use them in a similar lab assignment in the course COS126 - General Computer Science. The following is an excerpt from their description of the data contained in the file.

Each input file is a text file that contains information for a particular universe. The first value is an integer  $N$  which stores the number of bodies. The second value is a real number  $R$  which represents the *radius* of the universe: assume that all bodies will have coordinates that remain between  $-R$  and  $R$ . Finally, there are  $N$  rows and each row contains 6 pieces of data. The first two are the  $x$  and  $y$  coordinates of the initial position; the next ones are the  $x$  and  $y$  coordinates of the initial velocity; the next is the mass; and the last is a `String` that represents the name of an image file used to display the body. As an example, the input file `planets.txt` contains actual data for our solar system (and thus the name of our lab assignment) in MKS units.

```
5
2.50e11
0.000e00 0.000e00 0.000e00 0.000e00 1.989e30 sun.gif
5.790e10 0.000e00 0.000e00 4.790e04 3.302e23 mercury.gif
1.082e11 0.000e00 0.000e00 3.500e04 4.869e24 venus.gif
1.496e11 0.000e00 0.000e00 2.980e04 5.974e24 earth.gif
2.279e11 0.000e00 0.000e00 2.410e04 6.419e23 mars.gif
```

You will read the input file by using the object `FileIO` from the `myUtil` library. First, you will have to convert the external text data file into an instance of the Java `BufferedReader` object through the method `FileIO.openReader(...)`. Then, and only then, you can *read* data from the `BufferedReader` object in a similar manner to how you read data from the standard input. Please note that everytime that you try and access an external file, whether it is a text or an image or a sound file, you **have** to specify the full path to the file. For instance, the input data file `planets.txt` should be referred to inside your code as `H:/public_html/JAVA/ORF201/solar/planets.txt`. As an additional example, note the

usage of the variable `localPath` to load the background image file `starfield.jpg` inside `Solar.java`.

Note also that we are using the method `GL.getImage(...)` to convert an external image file onto a Java `Image` object. You will have to do the same with the images to be associated to each celestial body.

**7.3. Simulate the interaction among the bodies and draw their motion.** You will have to write the method `animate()` within which there is an infinite loop that repeatedly updates the position and the velocity of the bodies, and then draws them at their newly updated positions, by invoking the method `redraw()`. Please note the usage of the method `cFrame.pause(DELAY)` to control the *pace* of the simulation and thus the *rate* at which the frames are redrawn. *Do not forget to set the coordinate system of the drawing area appropriately so that the radius of the universe that you are simulating stays inside the window.*

With respect to the numerical part of the simulation, first you need to update the location of all of the bodies according to each body's velocity. This can be done using Newton's Second Law of Motion. Then, you need to update the velocities themselves. In order to be able to do this, you must first compute the acceleration of each body using Newton's Law of Gravitation. Then, you can use the Second Law of Motion again to update each body's velocity. For the data that you will be using in these simulations, use  $\Delta t = 25000$  for a reasonable trade-off between accuracy and computational effort.

By the way, as an optional nice touch, also suggested in the COS126 lab, you may play the theme to 2001 using the file `2001.mid` and the method `GL.playAudio(...)`.

For easier debugging, you should do the assignment in increments, stopping along the way to compile and check the correctness of your program. In fact, it is best to test your code first for two heavenly bodies only, and then test it for an array of multiple bodies.

## 8. MINIMUM REQUIREMENTS

Your program must:

- (1) implement a `Body` class containing all of the appropriate information about a heavenly body;
- (2) instantiate an array of `Body` instances that are specified by data read-in from an external input file;
- (3) correctly implement the model of gravitational interaction among bodies described in this assignment;
- (4) show the heavenly bodies moving around each other.

**Extra credit** (again, borrowed from COS126): Submit a universe created by you in the required input format, with the necessary image files. If its behavior is sufficiently interesting, you will get extra credit!