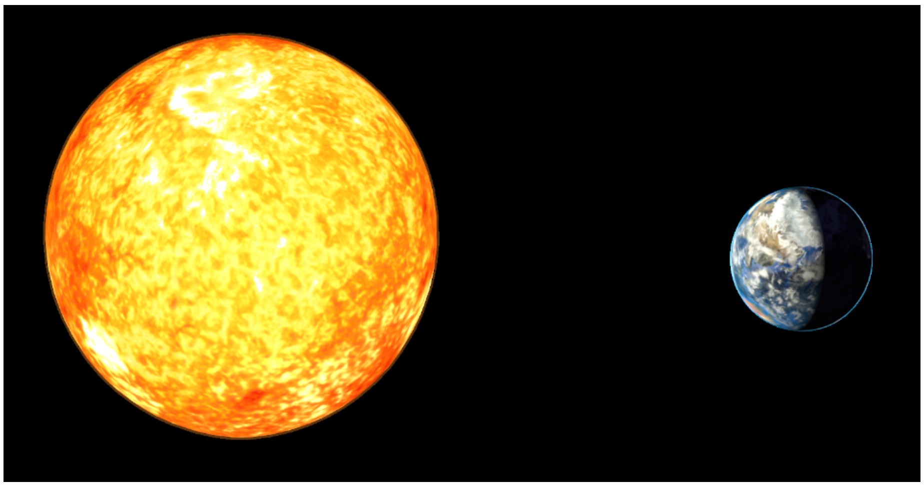
Laboratory 1 in Models and Simulation DD1354

January 23, 2015

1 Introduction



In this lab exercise you will explore the basics of physics simulation by implementing partial differential equations. This lies as a foundation to many of the physics simulations that you might have come across. The use cases range from games simple games, such as Super Mario, or more sophisticated games, such as Battlefield. Other applications are serious scientific applications, such as General Relativistic N-Body Simulations, and critical real-time systems, such as car engine software!

The lab exercise showcases a typical example: a solar system according to the classical equations of motion.

1.1 The Law of Universal Gravitation

The equations of motion are tools that describe properties of a physical system in mathematical terms. In general these are called *partial differential equations* and their solutions describe the evolution of certain physical properties, e.g. *position*, as a function of other *independent* physical quantities, e.g. *time*.

One of the most important and well known solutions to a *differential equation* comes from the study of planetary motion. It constitutes what we commonly call Newton's law of universal gravitation, which states:

"Any two bodies in the universe attract each other with a force that is *directly proportional* to the product of their masses and *inversely proportional* to the square of the distance between them."

$$\vec{F}_{ab} = -\vec{F}_{ba} = G\frac{m_a m_b}{d_{ab}^2}\vec{n}_{ab} \tag{1}$$

where $\vec{F}_{ab}$ represents the force that b exerts on a in the direction of the unitary vector $\vec{n}_{ab}$ that goes from a to b, and G is the universal gravitation constant. In this way, Equation 1 allows us to predict the evolution of a system of n masses $(m_1, \dots, m_n)$ merely knowing their initial positions and velocities.

1.1.1 A Historical Note

Tycho Brahe, in his quest for trying to disprove the heliocentric model gathered data about the position of the planets with respect to each other using primitive but rather precise methods.

Johannes Kepler, student of Tycho Brahe and follower of the heliocentric paradigm, analysed the data and made the following three observations that were consistent with Tycho Brahe's data:

1. The orbit of a planet is an ellipse with the Sun at one of the two foci.
2. A line segment joining a planet and the Sun sweeps out equal areas during equal intervals of time.
3. The square of the orbital period of a planet is proportional to the cube of the semi-major axis of its orbit.

Isaac Newton, inspired by the observations of Johannes Kepler, proposed the law of universal gravitation. It turned out that from such law he could mathematically derive the same three observations made by Kepler, but based on a theoretical analysis.

It is unclear if Newton arrived at the law of universal gravitation through solving the *partial differential equations* that represent Kepler's observations, or if he reached the law by trial and error.

Nevertheless, with this equation Newton managed to "Unite the heaven and the earth" under the same framework. It was due to the efforts of Tycho Brahe, Johannes Kepler and Isaac Newton that today we have a more complete understanding of the laws that govern the Universe.

1.2 What to do

To make things as clear as possible, the following is what you should do for this lab:

1. Go through section 2 (Dynamic Equations) and 3 (Simulation of the Solar System. Focus on the mathematics and how it is reflected in the code.
2. Install Blender. Look at section 4.1 (Setup) for details.
3. Solve the tasks in section 4.3 (Tasks). They involve both python programming as well as some work in Blender.
4. Submit your answers! See instructions at `https://www.kth.se/social/course/DD1354/page/lab-materials/`

2 Dynamic Equations

Consider a particle with mass m, initial position $\vec{p}_0$ and initial velocity $\vec{v}_0$. If this particle is under the effect of force $\vec{F}$ at time t it will experience an acceleration given by:

$$\vec{a}(t) = \frac{\vec{F}(t)}{m} \tag{2}$$

We know that velocity is the instantaneous variation of position per time unit, and that acceleration is the instantaneous variation of velocity per time unit. These two can be expressed as:

$$\vec{v}(t) = \frac{d\vec{p}(t)}{dt} \tag{3a}$$

$$\vec{a}(t) = \frac{d\vec{v}(t)}{dt} \tag{3b}$$

Integrating first Equation 3b, under the assumption that $\vec{a}$ is *locally* constant:

$$\vec{v}(t) = \int \vec{a}(t)\,\mathrm{d}t = \vec{a}t + \vec{v}_0 \tag{4}$$

And then using this value of $\vec{v}(t)$ to integrate Equation 3a

$$\vec{p}(t) = \int \vec{v}(t)\,\mathrm{d}t = \int \vec{a}t + \vec{v}_0\,\mathrm{d}t = \vec{a}\frac{t^2}{2} + \vec{v}_0 t + \vec{p}_0 \tag{5}$$

Which is the equation that allows us to calculate the position of a particle as a function of time given a known acceleration.

For the simplified planetary motion scenario, the law of universal gravitation dominates over smaller forces that could arise due to planet rotation and relativistic effects.

Assuming that the acceleration experienced by a planet is *locally* constant, i.e. has a negligible change with respect to the time step and other variables, one can make the following approximation for the position of planet i:

$$\vec{p}_i(t) \approx \frac{\vec{F}_i(t)}{m_i}\frac{t^2}{2} + \vec{v}_0 t + \vec{p}_0 \tag{6}$$

where $\vec{F}(t)$ is the total force experienced by the planet and m is its mass. However, we are interested in the general scenario: we have a set of n planets $\mathcal{P} = 1, \ldots, n$, where each of them is attracting the others according to the law of universal gravitation.

The total force acting on planet i is the sum of the forces that all other planets $j \neq i$ exert on it:

$$\vec{F}_i = \sum_{\forall j \neq i} \vec{F}_{ij} = \sum_{\forall j \neq i} G\frac{m_i m_j}{d_{ij}^2}\vec{n}_{ij} \tag{7}$$

where d_{ij} is distance between two planets and $\vec{n}_{ij}$ is the unitary vector between them:

$$d_{ij} = \|p_j - p_i\| = \sqrt{(p_j^x - p_i^x)^2 + (p_j^y - p_i^y)^2 + (p_j^z - p_i^z)^2} \tag{8a}$$

$$\vec{n}_{ij} = \frac{p_j - p_i}{\|p_j - p_i\|} = \left[\frac{(p_j^x - p_i^x)}{d_{ij}}, \frac{(p_j^y - p_i^y)}{d_{ij}}, \frac{(p_j^z - p_i^z)}{d_{ij}}\right] \tag{8b}$$

Combining equation 6 and 7 we get the expression for $\vec{p}_i(t)$:

$$\vec{p}_i(t < \Delta t) \approx \sum_{\forall j \neq i} G\frac{m_j \vec{n}_{ij}}{d_{ij}^2}\frac{t^2}{2} + \vec{v}_0 t + \vec{p}_0 \tag{9}$$

As planets begin moving, the acceleration will begin to change contradicting our assumption that it should be *locally* constant.

For this reason in order to make our computation valid we have to discretize time with enough granularity such that the acceleration does not change drastically from one sample to the next.

In practice, the values for $\vec{p}_i(t)$ given by Equation 9 will not be valid for all times t, but for a small window Δt during which the acceleration term, i.e. the summation accompanying $\frac{t^2}{2}$, is *locally* constant.

To obtain the position values for all times, Equation 9 is evaluated iteratively, i.e. using the result of step 1 as the initial condition of step 2, and so on. This means calculating $\vec{p}_i(\Delta t)$ using the initial $\vec{p}_0$ and $\vec{v}_0$ of planet i, updating them: $\vec{p}_0 \leftarrow \vec{p}_i(\Delta t)$, $\vec{v}_0 \leftarrow \vec{v}_i(\Delta t)$, and then repeating the process for another step Δt.

Notice that we have derived an equation that allows us to update $\vec{p}_0$ but not $\vec{v}_0$.

3 Simulation of the Solar System

The purpose of this lab is to use the law of universal gravitation in order to recreate the Solar System. To this end we will implement in Python a program that updates the position of the planets according to Newton's law. To display the results we will use Blender which is a widely used 3D rendering engine.

Even though the mathematical notation used to represent this procedure looks complicated, the coding part is much simpler. The code can be found in a separate file called **solar_system.py**, which is intended to be run from within the Blender scripting engine.

3.1 Code Overview

The following lines define libraries that are needed by the program: bpy is for interfacing with Blender primitives such as adding keyframes, moving objects, and interacting with the scene from Python; numpy is a mathematics library that supports vectors and many other functionalities.

```python
import bpy
from bpy import context
import numpy as np
```

The next lines of code defines the gravitational and time-stepping constants, as well as a class that will hold the information of each planet (i.e. name, mass, radius, position, acceleration etc.).

```python
G  = 6.674E-17 # Newton * Kmeter^2 / Kg^2
dt = 800.0

sun      = Planet(name = "Object_Sun", \
                  radius = 30*696342.00, \
                  mass = 1.98910E30, \
                  pos = np.array([0.0, 0.0, 0.0]), \
                  vel = np.array([0.0, 25*00000.0/1000, 0.0]))

earth    = Planet(name = "Object_Earth", \
                  radius = 300*006371.00, \
                  mass = 5.97219E24, \
                  pos  = np.array([0152098232.0, 0.0, 0.0]), \
                  vel = np.array([0.0, 25*29300.0/1000, 0.0]))

planets = [sun, earth]
```

The following lines define matrices that have as many rows and columns as planets are in the simulation. For each pair of planets, they store information about the forces (and direction of the force) they exert on each other.

```python
N = len(planets)
# magnitude of the force between 2 planets (every pair)
planet_force = np.zeros([N, N])
# vector of the direction between 2 planets (every pair)
planet_vector = np.zeros([N, N, 3])
```

The following lines loop through every planet i and calculate the forces that every other planet $j \neq i$ exert on it filling the corresponding cells in the matrices *planet_force* and *planet_vector*. Note how the formula for the universal law of gravitation is used and how the symmetry, $\vec{F}_{ij} = -\vec{F}_{ji}$, is reflected in the code.

```python
def calculate_forces():
    for i in range(N):
        for j in range(i):
            # force exerted from planet j
            planet_force[i][j] = (G * planets[i].mass * planets[j].mass) \
              / np.linalg.norm(planets[i].pos -   planets[j].pos)**2
            planet_vector[i][j] = planets[j].pos - planets[i].pos
            planet_vector[i][j] =  planet_vector[i][j] \
              / np.linalg.norm(planet_vector[i][j])

            # symmetrical
            planet_force[j][i]  =   planet_force[i][j]
            planet_vector[j][i] = - planet_vector[i][j]
```

The following lines we loop through every planet i and we sum up the forces that every other planet $j \neq i$ is exerting over it. With this data, we calculate the acceleration dividing by the mass, update the velocity and lastly update the position. The equations were already presented in the theoretical section.

```python
def update_pos_vel():
    for i, planet_i in enumerate(planets):
        force_total = np.zeros(3)
        for j, planet_j in enumerate(planets):
            force_total += planet_force[i][j] * planet_vector[i][j]

        planet_i.acc  = force_total / planet_i.mass
        planet_i.vel += planet_i.acc * dt
        planet_i.pos += 0.5 * planet_i.acc * dt**2 + planet_i.vel* dt
```

The following lines call the previously mentioned functions in the proper order and update the aforementioned variables frame by frame. For each frame that we wish to generate, the program calculates the forces between planets, updates their positions and then stores a keyframe that will be later rendered using Blender.

In order to present the result in blender, this code section will call the functions above to step forward in the simulation frame by frame. Each frame, a *keyframe* in blender is created which will animate the transforms of the objects in the blender scene. Note how the planet name is used to look-up the object in the scene. Why is the *scale* variable used?

```python
scale = 0.00000005
frame = 1;
n_frames = 270;

for t in range(1,n_frames):

    bpy.context.scene.frame_set(frame)
    frame += 1;

    calculate_forces()
    update_pos_vel()

    for planet in planets:
        planetObject = bpy.data.objects[planet.name]
        planetObject.select = True
        planetObject.location = [scale*planet.pos[0], \
                                 scale*planet.pos[1], \
                                 scale*planet.pos[2]]
        planetObject.rotation_euler.z = t/80

    bpy.ops.anim.keyframe_insert(type='LocRotScale', confirm_success=True);
```

4 The Lab

4.1 Setup

4.1.1 Files

The lab files can be found at: `https://www.kth.se/social/course/DD1354/page/lab-1-9/`

4.1.2 Windows

Blender
Download blender from `http://www.blender.org/download/` and install it.

Unity
Download unity from `http://unity3d.com/unity/download` and install it.

4.1.3 Mac

See windows.

4.1.4 Linux (Ubuntu)

Blender
If you are using a school computer it's already installed. If using your own computer it can be installed from a terminal with the command:

```
sudo apt-get install blender python3-numpy
```

Unity Unity does not run natively in linux. There are some workarounds using wine but it will not be described here.

4.2 Running The Code

You were provided with a Blender file called **lab1.blend**, it contains a scene with the Sun and the Earth, a Point Light and a Camera. Navigate to the *Text Editor* and you should see a script in Python that resembles the code described in the previous section.

Click on *Run Script*, this will run the simulation and generate the keyframes required to carry out the video rendering later. You can drag the cursor along the timeline to see how the planets move as a result of the keyframes generated when you ran the script.

Go into the *Camera Icon* in the *Properties Panel* and click on *Image*. You should see an image of the Sun and the Earth appear in the screen. Proceed to click on *Animation* to generate a video using the keyframes (Render → Run

Rendered Animation to play). For further reference on how to run the script refer to the Youtube video demonstrating its use.

4.3 Tasks

In this section you will be asked to modify certain aspects of the code to achieve different functionality. Doing this will give you insight into the details of the implementation and you will have a greater grasp of the potential of Blender as a rendering engine powered by the Python scripting language.

1. Study equation 9 in the end of section 2 and how it's derived. Provide the corresponding equation for the velocity update. [Hint: start with Equation 4 instead of Equation 5 like we did].
2. Try different values for dt in the simulation, for example 10, 100, 1000. Compare the results in terms of the trajectory that the planets generate. Why the trajectory generated with $dt = 10$ is shorter and more precise than the trajectory generated with $dt = 1000$?
3. Change the number of frames in your simulation n_frame to expand or shorten the length of the trajectory. Find a relation between the number of frames in your timeline and the duration of the video generated when you render it.
4. Change the stepping value between frames from $frames \mathrel{+}= 1$ to something like $frames \mathrel{+}= 10$. What happened to the keyframes generated, Why are they farther apart from each other? how does this affects the simulation[1] and video rendering? How does Blender fill in the gaps between frames 1 and 10.
5. Notice that the objects *Object_Earth_Atmo* and *Object_Earth_Cloud* follow the transformations of *Object_Earth*. How is this done?
6. Modify the code so that the Clouds do not rotate in the same direction of the Earth, but in the opposite direction and at half the speed.
7. In the code we are making the earth rotate around its z axis. In reality the Earth has a tilted axis of rotation of about 23 degrees called obliquity. Modify the code so reflect this and increase the realism of your simulation.
8. The planets in the simulation are set to be of arbitrary size. Modify your script so that it reads the actual radius of the planet from the variable $planet\_radius$ and scales the Sun and the Earth accordingly.
9. Modify the code so that the Sun slowly increases its size until it engulfs the Earth.
10. Open the file **planet_data.py** and use the information contained there to add another planet of your choice to the simulation. To add a new mesh to the scene you can create one via Add → Mesh → UV Sphere. You may also insert pre-made models if you find them online. Add textures to your new planet.
11. Select the camera, notice that the timeline is now empty because it has no keyframes attached to it. Move the camera to where you want to see the scene from, with the cursor on the 3D planel and the camera selected press 'i' to add a keyframe at the current frame indicated by the green bar in the timeline. Repeat the process to create an interesting animation for the camera.

Once you have managed to implement all these changes to the code, you will be able to make a very realistic rendering of the Solar System.

[1]remember to erase the previous keyframes using the *DopeSheet* in Blender

4.4 Submission

See instructions at `https://www.kth.se/social/course/DD1354/page/lab-materials/`

5 Resources

In the following section you will find link to useful Blender and Python resources that will help you to get through this lab successfully.

`http://www.blender.org/support/tutorials/` (version: 2.73a)
`https://docs.python.org/3.4/tutorial/`
`http://sun.turbosquid.com/`
`http://nasa3d.arc.nasa.gov/models`
`http://www.nasa.gov/multimedia/3d_resources/images.html`
`https://www.youtube.com/watch?v=6GLIbPKpgQ4`
`https://www.youtube.com/watch?v=n2n3WBDJTQI`

6 Troubleshooting

You will (most likely) need to: install python3 and python3 numpy, because Blender uses python3 and not python2.

sudo apt-get install python3-numpy

Copy the instalation of numpy for python3 into the blender directory:

sudo cp /usr/lib/python3/dist-packages/numpy /usr/lib/blender/scripts/modules/ -rf