Final Project: Planet Orbit Simulation

MATH 5061: Fundamentals of Computer Programming for Scientists and Engineers

Due: December 12th, 2016

Instructions

Please submit all your code as .py, .cpp and .h files and include the compiled binary program or output file. Create folders to separate the examples. Send them as a .tar.gz or ZIP file via email to math5061@temple.edu. Make sure to use the subject line (without quotes) “MATH5061:Assignment 0:ACCESSID” Where ACCESSID is your AccessNet ID, for example tue86537.

1 Python: Visualizing planet trajectories

The first part of this project is to write a Python program to create an animated plot of the output generated by the C++ part, which is a simulation of the orbits of planets for a given number of time steps. Example simulation output files are provided so that you can test the Python program without needing to complete the C++ part first.

You will implement the visualization using the third-party Python module Matplotlib which is popular for creating 2D plots. Using Matplotlib for animated plots was described in Thursday’s class.

As a guide, you could structure your program as follows:

- Create a class called `PlotTraj`. `PlotTraj` will read the structured input file and generate the animation for display on the screen.
- The `PlotTraj` class will have three methods, `__init__()`, `_update()` and `animate()` which are described below:

1.1 `__init__(self, filename)`

This method initializes the `PlotTraj` object by reading in data from the input file filename. The following data are contained in the input file

- Number of bodies
- Number of simulation steps or frames

- Names of the bodies
- Masses of the bodies
- Radii of the bodies
- Trajectory of the bodies which consists of one line per simulation step. Each line has the following space separated fields

  ```
  <step> <x1> <y1> <x2> <y2> <x3> <y3> ... <xn> <yn>
  ```

  `<step>:` simulation step

  `<x1>:` x coordinate of body 1

  `<y1>:` y coordinate of body 1

  `<xn>:` x coordinate of body n

  `<yn>:` n coordinate of body n

  `n:` number of bodies

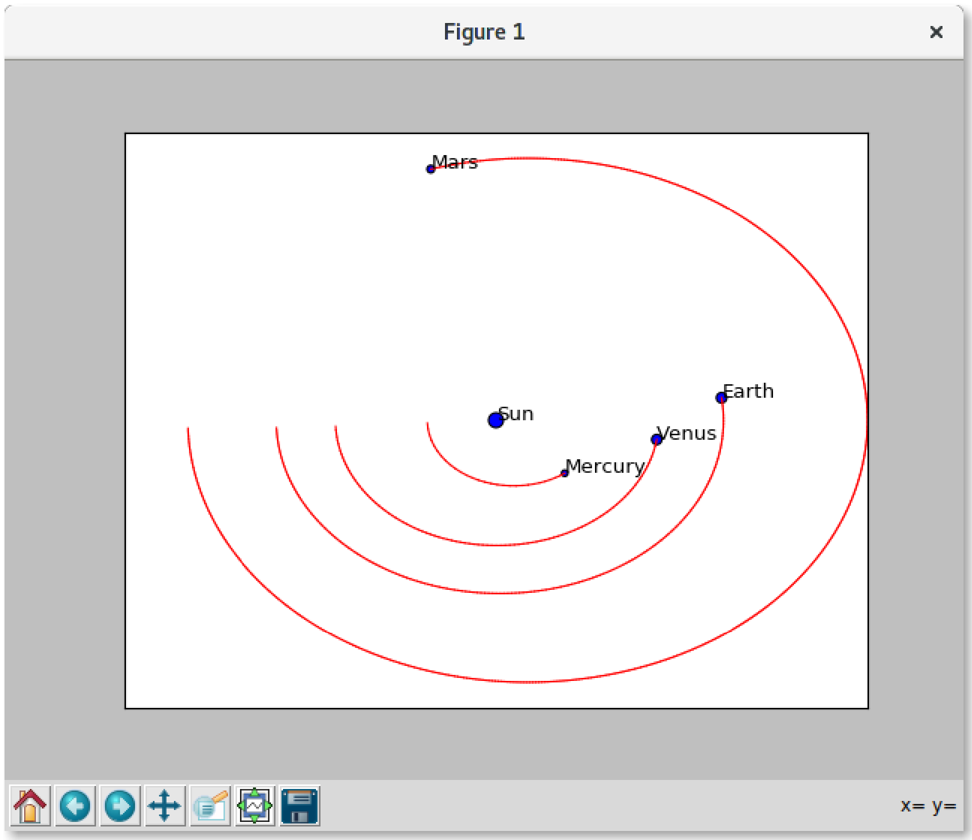
  For example, for a system with 5 bodies, each trajectory line will have 11 fields

Here are parts of an example file:

```
NUM_BODIES
2

NUM_STEPS
365

NAMES
Sun
Earth

MASSES
1.989e+30
5.972e+24

RADII
6.957e+08
6.371e+06

TRAJECTORIES
0 0.000000 0.000000 -147095000000.000000 0.000000
1 0.000000 0.000000 -147072101026.950928 -2617784148.577663
2 0.000000 0.000000 -147003411607.748016 -5234753260.311548
...
```

On the website you'll find three examples:

- `sun_earth.dump`
- `solar_system_core.dump`
- `full_solar_system.dump`



1.2 `_update(self, frame)`

This method receives an argument, frame, which is an integer that represents the current frame (timestep) that is to be plotted. `_update()` will update the positions of the various objects in the figure to reflect their state at time step frame.

1.3 `animate(self)`

This method will perform the actual plotting. It will create the figure, axes and scatter plots. It then creates a `FuncAnimation` object to setup the animation. Finally it will call the `show()` function of matplotlib to display the animation as described in class.

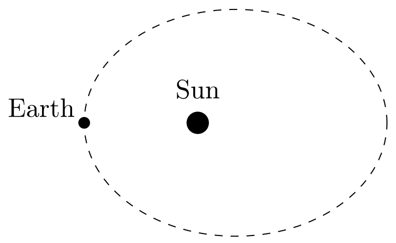
The final result of your program will be an animated plot that shows the planets as circles with labels and orbiting around the sun. Each circle's size will be proportional to its radius. You may use an arbitrary radius for the Sun since using its actual radius will make all the other objects too small in the plot. You should end up with something similar to the following which is a screenshot of an animation.

2 C++: Generating planet trajectories

The second part of the project is to write a C++ program which computes the orbits of objects in space, each interacting with each other through gravity. Because of the complexity of the code you will first write two simpler versions of it, which will lead up to the final program by gradually adding more complexity. It is important that you solve each task at a time and validate the functionality you have before proceeding. **When you submit your projects we want to have each version of the code, seperated into different folders.**

2.1 One planet moving around the sun

Your first version of the code will only simulate one single planet rotating around a fixed sun. Our simulations take place in 2D and we place our sun at the origin (0,0) of our coordinate system. Our planet, Earth, will be placed at (-147095000000, 0). This starting position is the closest our planet gets to our sun during a year.



To simulate how Earth moves around the sun we need to solve the equations of motion of the planet. In general, these equations are 2nd-order ordinary differential equations, one for each coordinate:

$$\ddot{x} = \frac{F_x}{m}$$
$$\ddot{y} = \frac{F_y}{m}$$

where F_x is the force acting on the planet in x-direction and F_y is the force acting in the y-direction. m is the mass of the planet. In order to solve the equations they are usually transformed into two 1st-order ordinary differential equations for each component:

$$\dot{x} = v_x$$
$$\dot{y} = v_y$$
$$\dot{v_x} = \frac{F_x}{m}$$
$$\dot{v_y} = \frac{F_y}{m}$$

where v_x is the velocity in x-direction, and v_y the velocity in y-direction of the object.

These type of equations can be solved numerically by a computer program. For this we discretize and integrate them. In essence how this works is take the current position (x_n, y_n) and velocity $(v_{x,n}, v_{y,n})$ of our planet and compute new values for each of these after a given time step Δt. We call these new values (x_{n+1}, y_{n+1}) and $(v_{x,n+1}, v_{y,n+1})$. As time step Δt we choose one 1 Earth day, which is 86400 seconds[1].

What our program will be doing is compute a new position and velocity of the planet after each day as illustrated in the following picture:

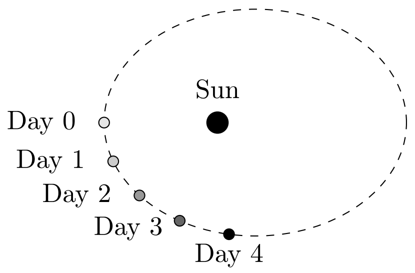


Figure 1: After each day we compute a new point of the planet's orbit

For this we need a formula to compute the new positions and velocities based on their current values and the time step:

$$\begin{aligned} x_{n+1} &= x_n + \Delta t(\ldots) \\ y_{n+1} &= y_n + \Delta t(\ldots) \end{aligned}$$

$$\begin{aligned} v_{x,n+1} &= v_{x,n} + \Delta t(\ldots) \\ v_{y,n+1} &= v_{y,n} + \Delta t(\ldots) \end{aligned}$$

2.1.1 Gravitational force between two masses

The force of one mass M of an object at location (x_M, y_M) acting on another object of mass m at location (x, y) is given by the following formulas in x- and y-direction.

$$\begin{aligned} \Delta x &= x_M - x \\ \Delta y &= y_M - y \\ F_x &= \frac{GMm}{\Delta x^2 + \Delta y^2} \frac{\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}} \\ F_y &= \frac{GMm}{\Delta x^2 + \Delta y^2} \frac{\Delta y}{\sqrt{\Delta x^2 + \Delta y^2}} \end{aligned}$$

where G is the gravitational constant:

$$G = 6.67408 \cdot 10^{-11}$$

[1]Note that for all following computations and formula it is important that you always use meters, kilogram and second values provided in this exercise.

With this formula and Newton's law $F = ma$, we can compute the amount of acceleration contributed by a mass M to a body at position (x, y):

$$a_x = \frac{GM}{\Delta x^2 + \Delta y^2} \frac{\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}}$$
$$a_y = \frac{GM}{\Delta x^2 + \Delta y^2} \frac{\Delta y}{\sqrt{\Delta x^2 + \Delta y^2}}$$

For our simplest example, where the sun is located at the orgin ($x_M = 0$, $y_M = 0$), the formula reduces to:

$$a_x = \frac{GM_{\text{Sun}}}{x^2 + y^2} \frac{-x}{\sqrt{x^2 + y^2}}$$
$$a_y = \frac{GM_{\text{Sun}}}{x^2 + y^2} \frac{-y}{\sqrt{x^2 + y^2}}$$

Note that in either case, a_x and a_y are functions of x and y:

$$a_x = a_x(x, y)$$
$$a_y = a_y(x, y)$$

So depending where you are relative to the body with mass M you experience a different acceleration due to the gravitational force.

2.1.2 `Body` class

Write a class named `Body` for objects in space. Each object should store the following attributes:

- name (use `std::string`)
- position (x, y)
- velocity (v_x, v_y)
- mass M
- and radius R.

Add a constructor to the class which allows you initialize all of these. Finally write a member function called `compute_acceleration` which computes the acceleration induced by this body on an object at location (x, y).

```
class Body {
public:
    ...
    void compute_acceleration(double x, double y,
                              double & ax, double & ay);
};
```

Verify that the following code produces the same results with your `Body` class:

```
const double SUN_MASS = 1.989e30;
const double SUN_RADIUS = 695700000.0;
// x, y, vx, vy of sun are all 0

Body sun("Sun", 0, 0, 0, 0, SUN_MASS, SUN_RADIUS);
double ax, ay;
sun.compute_acceleration(1000000000.0, 500000000.0, ax, ay);
printf("ax: %f\n", ax); // ax: -94.986344
printf("ay: %f\n", ay); // ay: -47.493172
```

2.1.3 Integrating a single time step

Next you will write the code necessary to integrate one single time step and compute the new position and velocity of your planet. The goal is to find the new location of Earth after one day. Here is the program you should use to test your code. Next we'll explain how to implement the `step` method.

Listing 1: Simulate single day step of Earth

```
const double SUN_MASS = 1.989e30;
const double SUN_RADIUS = 695700000.0;
const double EARTH_X0   = -147095000000.0;
const double EARTH_Y0   = 0.0;
const double EARTH_VX0  = 0.0;
const double EARTH_VY0  = -30300.0;
const double EARTH_MASS = 5.972e24;
const double EARTH_RADIUS = 6371000.0;
const double dt = 86400.0; // 1 earth day in seconds

Body sun("Sun", 0, 0, 0, 0, SUN_MASS, SUN_RADIUS);
Body earth("Earth", EARTH_X0, EARTH_Y0,
                    EARTH_VX0, EARTH_VY0,
                    EARTH_MASS, EARTH_RADIUS);

earth.step(dt, &sun); // TODO, explained next

                              // this should output:
printf("x: %f\n", earth.x);   // x: -147072101026.950928
printf("y: %f\n", earth.y);   // y: -2617784148.577663
printf("vx: %f\n", earth.vx); // vx: 530.054352
printf("vy: %f\n", earth.vy); // vy: -30295.283069
```

2.1.4 Numeric integration using a 4th-order Runge-Kutta integrator

The numeric integration of the new position and velocity are the heart of our orbit simulation program. It should be implemented in a `step` member function in the `Body` class. The method takes two parameters: the time step `dt` and the body which creates forces. Right now the only body doing this is the sun, so we pass the sun body object as a pointer to this function. Inside we will use the `compute_acceleration` method of the sun object.

```
class Body {
...
    void step(double dt, Body * sun) {
        ...
    }
};
```

Because we are using such a large time step, we have to write the integration using a very sophisticated method called a Runge-Kutta integrator of 4th order. While it all looks and sounds scary, we will provide you the recipe of how to implement it. You don't have to understand where it came from or why it does what it does. All you need to know is that its purpose is to compute the new position and velocity using the following scheme and translate the formulas into code:

$$x_{n+1} = x_n + \frac{\Delta t}{6}\left(k_{1,x} + 2k_{2,x} + 2k_{3,x} + k_{4,x}\right)$$
$$y_{n+1} = y_n + \frac{\Delta t}{6}\left(k_{1,y} + 2k_{2,y} + 2k_{3,y} + k_{4,y}\right)$$

$$v_{x,n+1} = v_{x,n} + \frac{\Delta t}{6}\left(k_{1,v_x} + 2k_{2,v_x} + 2k_{3,v_x} + k_{4,v_x}\right)$$
$$v_{y,n+1} = v_{y,n} + \frac{\Delta t}{6}\left(k_{1,v_y} + 2k_{2,v_y} + 2k_{3,v_y} + k_{4,v_y}\right)$$

As you can see the new position (x_{n+1}, y_{n+1}) and velocities $(v_{x,n+1}, v_{y,n+1})$ are dependent on the time step Δt, the current position (x_n, y_n) and velocity $(v_{x,n}, v_{y,n})$, as well as 16 parameters ($k_{1,x}$, $k_{2,x}$, $k_{3,x}$, $k_{4,x}$, $k_{1,y}$, $k_{2,y}$, $k_{3,y}$, $k_{4,y}$, k_{1,v_x}, k_{2,v_x}, k_{3,v_x}, k_{4,v_x}, k_{1,v_y}, k_{2,v_y}, k_{3,v_y}, k_{4,v_y}).

The tricky part is computing these parameters, but all you have to do is follow the next steps and combine them to the above equation to get the new positions and velocities.

1. Compute $k_{1,x}$ and $k_{1,y}$:

$$k_{1,x} = v_{x,n}$$
$$k_{1,y} = v_{y,n}$$

2. Compute k_{1,v_x} and k_{1,v_y}:

$$\begin{aligned} k_{1,v_x} &= a_x(x_n, y_n) \\ k_{1,v_y} &= a_y(x_n, y_n) \end{aligned}$$

   Note this is done by calling the `compute_acceleration` function of the sun body object.

```
double k1vx, double k1vy;
sun->compute_acceleration(x, y, k1vx, k1vy);
```

   All following steps which evaluate a_x and a_y do the same, but use different x and y values.

3. Compute $k_{2,x}$ and $k_{2,y}$:

$$\begin{aligned} k_{2,x} &= v_{x,n} + \frac{\Delta t}{2} k_{1,v_x} \\ k_{2,y} &= v_{y,n} + \frac{\Delta t}{2} k_{1,v_y} \end{aligned}$$

4. Compute k_{2,v_x} and k_{2,v_y}:

$$\begin{aligned} k_{2,v_x} &= a_x(x_n + \frac{\Delta t}{2} k_{1,x}, y_n + \frac{\Delta t}{2} k_{1,y}) \\ k_{2,v_y} &= a_y(x_n + \frac{\Delta t}{2} k_{1,x}, y_n + \frac{\Delta t}{2} k_{1,y}) \end{aligned}$$

5. Compute $k_{3,x}$ and $k_{3,y}$:

$$\begin{aligned} k_{3,x} &= v_{x,n} + \frac{\Delta t}{2} k_{2,v_x} \\ k_{3,y} &= v_{y,n} + \frac{\Delta t}{2} k_{2,v_y} \end{aligned}$$

6. Compute k_{3,v_x} and k_{3,v_y}:

$$\begin{aligned} k_{3,v_x} &= a_x(x_n + \frac{\Delta t}{2} k_{2,x}, y_n + \frac{\Delta t}{2} k_{2,y}) \\ k_{3,v_y} &= a_y(x_n + \frac{\Delta t}{2} k_{2,x}, y_n + \frac{\Delta t}{2} k_{2,y}) \end{aligned}$$

7. Compute $k_{4,x}$ and $k_{4,y}$:

$$\begin{aligned} k_{4,x} &= v_{x,n} + \Delta t \cdot k_{3,v_x} \\ k_{4,y} &= v_{y,n} + \Delta t \cdot k_{3,v_y} \end{aligned}$$

8. Compute k_{4,v_x} and k_{4,v_y}:

$$\begin{aligned} k_{4,v_x} &= a_x(x_n + \Delta t \cdot k_{3,x}, y_n + \Delta t \cdot k_{3,y}) \\ k_{4,v_y} &= a_y(x_n + \Delta t \cdot k_{3,x}, y_n + \Delta t \cdot k_{3,y}) \end{aligned}$$

Once you have implemented this function, verify that the above earth test program gives you the same answer.

2.1.5 Simulate one earth year and output trajectory

Expand the test earth test program and simulate 365 days. Generate a text output file using `fopen`, `fprintf` and `fclose` which is compatible with your Python visualization.

```
// output header information (NUM_BODIES, NUM_STEPS,
// NAMES, MASSES, RADII)

// output initial position (time step 0)

for(int nstep = 1; nstep <= 365; nsteps++) {
    // output current position
    ...
    earth.step(dt, &sun);
}
```

Here is a part of this output:

```
NUM_BODIES
2

NUM_STEPS
365

NAMES
Sun
Earth

MASSES
1.989e+30
5.972e+24

RADII
6.957e+08
6.371e+06

TRAJECTORIES
0 0.000000 0.000000 -147095000000.000000 0.000000
1 0.000000 0.000000 -147072101026.950928 -2617784148.577663
2 0.000000 0.000000 -147003411607.748016 -5234753260.311548
3 0.000000 0.000000 -146888954238.876648 -7850092591.654999
4 0.000000 0.000000 -146728766403.907928 -10462987985.495131
5 0.000000 0.000000 -146522900557.649872 -13072626163.974415
6 0.000000 0.000000 -146271424103.975403 -15678195020.838345
7 0.000000 0.000000 -145974419367.343323 -18278883913.151958
...
```

2.2 Multiple planets (not interacting with eath other)

Extend your program to allow simulating multiple planets at the same time. However, continue to keep it simple by only computing the acceleration due to the sun. Keep a list of objects in your simulation, either as `std::vector<Body*>` or array of `Body` pointers and use it shorten your program using loops.

Adjust the output of your program to include the trajectories these planets. Here are some more initial parameters for planets our solar system:

Name	x_0	y_0	$v_{x,0}$	$v_{y,0}$	mass	radius
Sun	0	0	0	0	$1.989 \cdot 10^{30}$	695700000
Mercury	-46000000000	0	0	-58980	$0.33011 \cdot 10^{24}$	2439700
Venus	-107480000000	0	0	-35260	$4.8675 \cdot 10^{24}$	6051800
Earth	-147095000000	0	0	-30300	$5.972 \cdot 10^{24}$	6371000
Mars	-206620000000	0	0	-26500	$6.4171 \cdot 10^{23}$	3389500
Jupiter	-740520000000	0	0	-13720	$1898.19 \cdot 10^{24}$	71492000
Saturn	-1352550000000	0	0	-10180	$568.34 \cdot 10^{24}$	54364000
Uranus	-2741300000000	0	0	-7110	$86.813 \cdot 10^{24}$	24973000
Neptune	-4444450000000	0	0	-5500	$102.413 \cdot 10^{24}$	24341000

Table 1: A set of initial parameters for the orbits in our solor system. Note that lenghts and distances are in meters, velocities in meters/second and mass in kilogram.

2.3 Multiple planets (interacting with eath other)

At this point you should be have a running simulation of multiple planets and a working output which you can visualize using your Python program. The final version of the C++ program should now also include the forces other planets have on each other.

This will require two major changes: (1) first the `step` function should now get all bodies in your simulation as second parameter. Not only the sun. (2) the total acceleration is now the sum of all contributions by all the bodies in the simulation. This means you have to adjust all the places where the acceleration function is used.

$$a_{x,j,total}(x,y) = \sum_{i \neq j} a_{x,i}(x,y)$$
$$a_{y,j,total}(x,y) = \sum_{i \neq j} a_{y,i}(x,y)$$

where $a_{x,i}(x,y)$ and $a_{y,i}(x,y)$ are the x- and y-acceleration contributions of body i on body j.