Contents

1 Introduction

So, why did I write this program, and what does it do, anyway?

I had a brilliant idea for a science-fiction story, and then I realized that it depended strongly on the orbital mechanics of the solar system that I was constructing. Since I wanted to get this right, I set off on a brief quest across the Internet for a suitable program that let me plug in my own planets, how fast they were going, how large the star was, and then run time forward to see how everything moves. Unfortunately, I didn't quite find what I was looking for – there are some simple ones for our own solar system, and a cute one associated with the Kerbal Space Program, and some extremely simplistic models that are only qualitative, lacking relevant details like any means for estimating close approaches between planets.

Thus, I gave up and decided I would write the thing I wanted.

It's not done yet. I ultimately plan to write up a nifty GUI for all of this, to make it easier and more intuitive to use. And perhaps I'll put it up as a web application, if I'm feeling ambitious.

At present, it is less user-friendly than my ultimate goal. It is currently a small N-body solver which runs on the command line, and spits out as much information as you want.

2 Installation

Here are the handy installation instructions. These are geared towards someone with at least a basic familiarity with the Terminal application on Mac/Linux, and the equivalent on Windows. If you need an introduction to working on a command line, you should read up on that quickly elsewhere. (Don't worry; it's less intimidating than it sounds.)

2.1 Mac/Linux Installation

Open up a terminal, and navigate to the directory that contains the downloaded files. (You should see a README file in this directory.)

At the terminal, run:

```
make
```

Yes, that's it. This should run with no warnings, and produce an executable file named SolarSystem.

2.2 Windows Installation

I have not yet determined how to perform Windows installation, although it . This is on the to-do list.

3 General Usage

Once you've make the executable file, the format for for running the program is one of the following:

```
./SolarSystem --help
./SolarSystem parameter_file [options]
```

The first command will print the quick help description . Using the "–help" option will always print the help description, and will override any other input options.

At a minimum, you will need to run with an input parameter file (here labeled parameter_file). All the other inputs are optional. The optional inputs have the following effects:

- -f: This specifies the input file format. By default, this is 0, which means that inputs files must have the positions and velocities of each object in the system. If you set this to 1, it will instead take in the orbital elements for each object. More on the parameter file details is given in § 3.1.
- -t: This is how much time there is between timesteps, in seconds. By default, this is 3600 s (one hour). For the example parameter files, I recommend a half-hour integration time (1800 s). Otherwise, you will get significant numerical error for the objects with very short orbital periods.
- -r: How long to run, in Earth-years. By default, this is ten years. You'll want to run longer if you want to see Neptune orbit the sun once, and shorter if you're interested in looking at just an orbit or two or Earth.
- -s: How many timesteps between printouts. This defines how often . By default, every timestep is printed. If you enter the option "-s 10", every tenth timestep will be printed out, starting with the first one. If you have a half-hour timestep, this means that you know where the objects are with

five-hour spacings. Printing out fewer timesteps will speed up how quickly SolarSystem runs, at the cost of not having as much information.

- –minimalhalt: By default, a calculation is run at every timestep to determine if two objects that are close to each other are on a collision course. If a collision is found, the program stops running. Entering this option, you turn off this extra calculation. Be warned – while the program will still stop if there's a collision, some collisions will be missed this way, especially if your timestep is large.

- –norecenter: By default, the program automatically shifts the coordinates and velocities of all the objects so that they don't drift away from the zero of the coordinate system. Entering this option turns that feature off.

For instance, a sample command that you can run with the files I've already provided might be:

```
./SolarSystem data/solarsystem.dat -o data -r 20 -t 1800 -s 2
```

This will run with the initial conditions provided in the data/solarsystem.dat file, and output then results to the data directory. The program will calculate the motions of the Sun and all the planets for a period of 20 years, with half-hour timesteps, but only printing out the results with one-hour spacing (every 2 timesteps). Go ahead and run this – it should go fairly quickly, and some diagnostic information on how it runs will being printed to your terminal. There will be a few output files, with names like "Mars.dat". Each file lists the positions and velocities of each objects, in the x, y, and z directions, at each output timestep. These will be in units of meters and meters per second.

3.1 Parameter Files

Here's a description of what needs to be in that parameter file, which is the meat of your input. For solving any gravitational N-body problem, you have to know what particles (stars, planets, etc.) you're starting with. You need to know their mass, and their positions and velocities in all three spatial dimensions. These are all given in the input parameter file, along with a name and radius for each object.

First, I do not recommend using the orbital element input format (-f 1). The orbital elements are converted to position and velocity coordinates for the integration, and this is done assuming that the orbital elements are for objects orbiting the Sun. It won't work for any other system, and isn't as intuitive to modify. Thus, most of my discussion will be for the default format (-f 0).

Each line of the input parameter file should look something like this:

```
Sol 1.988544e30 6.955e8 1.8e8 -3.4e8 -1.39e7 0 0 0
```

Each line should look like this. The first entry should be a name for the object, with no spaces. ("TheEarth" is fine, but "The Earth" is not.) The name you choose will be the preface of the The second entry should be the

object's mass, in kilograms. (Yes, the Sun is that massive.) The second entry should be the object's radius in meters. The next three entries should be the x, y and z coordinates of the object at the initial time, also in meters. The remaining three inputs are the velocity of the object in each of the x, y, z directions, in meters per second.

If you decide you don't want to include an object any more, just put the comment character "#" in front of the line, and it will be ignored.

You can set the radius for an object to zero. In that case, it's treating as an infinitesimally small particle with finite mass, and will not collide with other objects that also have a radius of zero.

You can also set the mass for an object to zero. In that case, you're assuming that the object is so small that it exerts no gravitational pull on anything else. This is only reasonable to do if you're modeling, for instance, a spacecraft flying through the system.

I have included two example input parameter files in the data directory.

The first is named solarsystem.dat. It includes the Sun, eight planets, and the Moon, all in the appropriate starting positions for January 1, 2014. If you would like to test how well my program performs, you can compare the outputs after running with the positions given by JPL's Horizons database (`http://ssd.jpl.nasa.gov/?horizons`). You can also use that database if you want to get input values for other objects in our solar system (like, say, Pluto).

The second parameter file is named jackpot.dat. This is a fictional solar system, which I'm planning on using in a novel. It's a system with five terrestrial planets, and a star a little smaller than the Sun. If you run this system for long enough... something interesting happens. Have fun!

If you would like to make your own solar system from scratch, here are a few things to consider. First, if you're working with a single central star, it should be nearly at rest at the center of everything.

Second, you can try just putting in a planet 1 AU away (about $1.49597870 \times 10^{11}$ meters) with zero velocity. You won't have to wait very long before it falls directly into the star. In order to place an object into a stable circular orbit around a much more massive object, you'll need to give it some velocity in a direction perpendicular to the direction from the planet to the star. This velocity is given by:

$$v = \sqrt{\frac{GM_\odot}{R}} \tag{1}$$

$M_\odot$ is the mass of your central star (or whatever other object), in kg. R is the distance between the massive central object and the object you're adding, in meters. G is Newton's constant, which has a value of 6.67384×10^{-11}N m^3/kg/s^2. If the two objects orbiting each other are of similar mass, you'll need to do something a little more complicated, since both object should have a nonzero velocity. But this will still get you close if one is much more massive.

Speaking of which, even in our solar system, the planets will tug on the star, giving it a little bit of velocity, and offsetting the star from the center of mass

of the system. If you don't want to have to calculate that, there's a way to cheat: set up the system with your star at the center, then run the program recentering on. Now, look at the first line of the star's output. The recentering has been done for you. Take the new positions and velocities and plug them in for the central star, without changing the values for any of the planets. That should get you pretty close.

Finally, do note that over long times (if you run for hundreds of years, say), small differences can add up. A difference in position of less than a meter can determine whether an asteroid hits the Earth or not in a few millennia. If you make a chaotic system, like the one in my Jackpot system, try fiddling a little bit with the parameters to see how it changes the results.

If you want to make a binary star system, that's a bit more complicated, but Wikipedia is a pretty good resource for getting a handle on how to set that up. Do the binary stars first, and make sure your two stars orbit each other in a stable fashion before you start throwing planets in there.

4 Specifications and Performance

And here are some gory details.

The program is written in C++.

For runs on my MacBook Pro laptop computer, running with 9 objects takes about five minutes per hundred years with a timestep of 1800 s and printing out once per hundred timesteps.

The first few integrations are performed using a fourth-order Runge-Kutta algorithm. After that setup is done, all further timesteps are calculated using a fourth-order predictor-corrector which combines an Adams-Bashforth prediction step with an Adams-Moulton correction step. If you want to play with writing something of your own, I have included the Runge-Kutta and predictor-corrector algorithms in a couple of separate header files for your entertainment. Be aware that over long times, if you use only the Runge-Kutta integrator, it will have rather severe numerical drift problems.

All gravity calculations are done in a purely Newtonian framework. Collision checking is performed whenever the centers of two objects are within twenty times the sum of their radii, and are done under the two-body assumption for the two objects in question.

5 Caveat Emptor

This is the list of all the known issues and nitty-gritty details about the SolarSystem software. The algorithm I'm currently using is far from perfect, and it's best that you know where and how it can break. Some of these are planned improvements, and where that's the case, I have made a note indicating such.

If you find something I haven't noted here, then please let me know!

1. At present, I do not have this set up with easy Windows compatibility. If you want to, you can write your own NMAKE file, and compile it yourself. However, I do plan on writing my own version of this and testing on the WIndows partition of my computer.

2. This is a direct N-body solver. As a consequence, it has to calculate the force between all pairs of particles. The resulting algorithm is $O(N^2)$. If you double the number of objects you wish to track, the amount of time to run will increase by a factor of four. While there are methods for speeding this up, they all result in what I consider to be an undesirable loss of accuracy, and are not terribly useful for the small number of objects I've been working with.

3. I do not include the effects of any non-point particles. For instance, I don't include any calculations of tidal interactions, or the gravitational effects of disks or clouds.

4. I do not include any corrections to deal with the fact that gravity is limited by the speed of light. (Earth feels the Sun where it was eight minutes ago, not where it is now.) This is a significant correction, and I think the largest source of error at the present time. I plan an update that corrects for this effect.

5. I do not include any relativistic or post-Newtonian gravity corrections (i.e., general relativity). This is important to, for instance, correctly calculating the precession of Mercury's orbit. I do not plan to add this improvement at the present time.

6. Unit conversions are currently not included. While I do plan on including a unit conversion section, which would allow the input positions in the parameter file to be input as AU (for instance), it's a low priority.

7. I make no attempt at modeling collisions between planets or stars after the point of contact. Seriously, people spend their entire careers doing things like modeling the moon-forming impact and planet formation and asteroid collisions. It's messy, and goes way beyond gravity. That's why this program stops whenever it hits a collision, since it doesn't know what to do with them. I'm not going to include this. Ever. Sorry.