Benjamin Åkerlund
665762 ELEC/EST

Y2: Programming project, Solar system simulator

General description:

For the final Y2 course project I have created a Solar system simulator. The program simulates the movement of heavenly bodies (planets) and satellites by measuring points of their orbits, with a frequency of 1 measurement per elapsed day, until either a collision happens or the simulation time runs out.

The most important contributing factors in this sim are the effects of gravity on each body and the Newtonian laws of physics. The simulation does not account for the effects of eg. air resistance. The main attributes of each body will be velocity, location, size (radius) and mass. For this I have implemented a working 3D coordinate system and a graphical interface to allow the user to view the simulation as it runs.

The user may control the duration (days) and the speed (how many days pass per second) of the simulation. In addition to this the user may add a satellite to the simulation, and then by running it see how it would behave in the system. The user can control the satellites starting location, starting velocity and mass.

The simulation upon first being started, requires the user to load a scenario before anything else can be done. This means that the user has the freedom to load a desired scenario eg. based on a certain date and time or even one that is different to the Solar System we all live in. I have included a ready save file called INIT_8.txt which contains the data of the planets in our Solar system (on 01-01-2020 00:00). However, any save file (or scenario eg. made by the user) with this specific structure may be loaded into the simulation. I have also included a Savefile_structure_plan.txt to closer explain the save file structure.

I have made the project according to the intermediate difficulty level with the graphical user interface as a main focus.

Instructions for the user:

By first launching the program by running the main.py file the user is greeted with a UI that consists of an empty black screen and on the right side a column to input parameters. To load a situation the user should click the “Load” button, which opens a filedialog box from where the user may select the wished save file to load (eg. INIT_8.txt). The ready made save files are located in the Savedata folder. After

selecting a file, the planets should now appear in the black window and the scenario is now loaded.

To then run the simulation the user must first input the duration of the simulation (in days) as well as the speed (how many days will pass per second of real time). After these are typed into the corresponding fields the simulation can be run by clicking the “Run” button. If the user wishes the simulation may also be stopped by clicking the “Stop” button. If the simulation stops the user may click “Run” again to resume the simulation.

In addition to this the user may also load a satellite into the simulation. By inputting the satellite parameters after the scenario is loaded, and finally clicking the “Load Sat” button a satellite should appear as a white ball. If the user now runs the simulation, they should be able to see how the loaded satellite behaves in the solar system.

External libraries

When creating the ready made INIT save file I used the astropy external library to acquire all of the planetary data. However, this library is not required to be installed for the user to be able to run the program. Other than that the only external library used in the program is the PyQt5 library. This is required to be installed in order to run the program.

To help teach me how to create a User Interface I experimented a bit with the QtDesigner tool, that lets you create a UI by dragging and editing properties of eg buttons and text labels and finally outputting this into ready code that you can run. I ended up not using anything generated by this tool for the program source code, since the Designer generated code is awfully messy and inefficient. Nevertheless, this was a good source for learning the basics.

Also the math library was used.

Structure of the program

The most central classes for the backend of the program are Planet and World (world.py and planet.py). These are the backbone of the whole program. The World class describes the world (or solar system) that all of the heavenly bodies are stored in as a list of Planet objects. The World class contains all of the physics algorithms necessary for the simulation to work. These algorithms are explained in more detail below. The most essential method in the World class is the update_all() method. This

method calls the physics algorithms and updates the location and velocity of each planet object that inhabits the World.

The heavenly bodies are Planet objects with their main attributes being mass, radius, location and velocity. This class contains methods to set all of these attributes for a Planet object. It also contains the conversion of units from eg. [km] to [m] or from [km/d] to [m/s]. These methods are called when the LoadSaveIO class reads through the save file. I created the LoadSaveIO class in order to create all of the planet objects with all of their corresponding attributes based on the data of a save file. The class method load_save is called when the user wishes to load a scenario, after which it reads through the chosen text file and adds the data into the World. This class was modeled after the HumanWriteableIO class from the course assignments (Week 5) in order to make creating save files as user friendly as possible. It should be able to read any save file, as long as the file is in the format I have specified. More about this in the Savefile_structure_plan.txt file.

Finally on the frontend of the program are the UI_MainWindow and PlanetGraphicsItem classes (GUI.py and planet_graphics_item.py). The UI_MainWindow class creates a graphical user interface and is called when running main.py. This class contains all of the methods needed for the visualisation of the simulation. Most importantly it is divided into methods for first initialising different parts of the UI. The input parameters are initialised in the init_input() method, which creates the group box inside which all of the input and user controls are set. The init_grapics() then shortly creates a graphics scene and a view into the scene (the big black box). Into this scene all of the graphics items are loaded by the different methods in this class.

The most significant methods in the UI_MainWindow class other than these are the add_planet_graphics_items() and the update_graphics_items() methods. These methods are called by different input_handlers (eg. Load_handler) based on commands given by the user. The add_ method is accessed once when loading the scenario into the simulation (“Load” button). After this method adds all of the initial graphics items, the update_ method keeps updating the positions of these items (“Run” button) based on their positions in time. Also when the satellite is loaded it is created in a similar way, and the update_ method also updates this graphics item.

The PlanetGraphicsItem class inherits the attributes of a QGraphicsEllipseItem and contains the data of how every graphical item in the scene is to be drawn. Most importantly this class contains a method to convert the “color” of a planet from what is stored in the save file to an actual color inside the QColor() library. This lets the program draw a planet object even if the color is not defined in the QColor() library,

and by further developing this method a larger range of colors may be supported in the program.

Algorithms

- **calc_force(p)**
  This algorithm is the most central one in the backend of the program. It works to calculate the total force exerted onto a specific planet object “p” by all of the other planet objects in the whole solar system. It iterates through the list of planets stored in the world and compares the “p” planet to it in order not to calculate a planet's own force on itself which would end with a catastrophic division by 0. All of the units are according to SI standards which means they are already converted before this. The algorithm adds a force to the total sum force of “p” in each dimension (x, y, z) and finally sets the total sum force exerted on “p” as a “negative”. This is done since the reference point of view for this calculation is that a force acting “outwards” from the planet is negative, and “inwards” is positive.

  The algorithm uses the newtonian law of gravity to calculate the force but adds a factor of the distance between the objects in each dimension (eg. dx) and divides by the total distance once more to even this out. This is in order for the vector calculus to be executed correctly.

$$F = G\frac{m_1 m_2}{r^2} \qquad d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

```
G = float(6.675 * 10.0**-11.0)
fx = 0
fy = 0
fz = 0

for planet in self._planets:
    if not planet == p:
        dx = (p.x_loca - planet.x_loca)
        dy = (p.y_loca - planet.y_loca)
        dz = (p.z_loca - planet.z_loca)
        d = sqrt(dx**2.0 + dy**2.0 + dz**2.0)

        fx += dx * (G * planet.mass * p.mass) / (d ** 3.0)
        fy += dy * (G * planet.mass * p.mass) / (d ** 3.0)
        fz += dz * (G * planet.mass * p.mass) / (d ** 3.0)
p.set_force(-fx, -fy, -fz)
```

Benjamin Åkerlund
665762 ELEC/EST

- **update_vel(p)**
  This algorithm uses the already calculated sum of forces on a planet “p” (in each dimension). Thanks to Newton's second law of motion the calculated force can be converted into acceleration since F=ma. Now with a=F/m we can calculate the new velocity of a planet “p” after these forces have been exerted on it during a period of t = 1d. The formula used is the special case of constant acceleration, depicted below. This formula is perhaps not totally accurate, since it accounts for the acceleration being exerted on a planet as being constant. This is true in the case of this program since the program calculates the forces on a planet each day and then sets the velocity as constant for a whole day until the next calculation. This may not be entirely accurate, but on such a large scale as the universe I feel it is accurate enough for the purpose of this simulation.

  A more accurate algorithm could be defined by calculating eg. for each passed hour or minute or second, but when the simulation time usually is around hundreds of days or even years, it would cause a significant load on processing.

  $$\boldsymbol{v} = \boldsymbol{u} + \boldsymbol{a}t$$

  with $\boldsymbol{v}$ as the velocity at time t and $\boldsymbol{u}$ as the velocity at time $t = 0$.

```
t = float(60*60*24)
vx = (p.x_vel) + ((p.fx / p.mass) * t)
vy = (p.y_vel) + ((p.fy / p.mass) * t)
vz = (p.z_vel) + ((p.fz / p.mass) * t)
p.set_velocity(vx, vy, vz)
```

- **update_loca(p)**
  This algorithm is the final one called when updating the data of each planet after a time period of one day. The algorithm calculates the new position of a planet “p” by solving each one of its coordinates in the x,y,z dimensions. The formula used for this is x = x0 + v*t, in which x0 is the initial coordinate, x is the new coordinate and v is the velocity. T again is time and is set to be 1d. The algorithm does this calculation for the planet in each dimension and sets the final x,y,z coordinates of the planet.

Benjamin Åkerlund
665762 ELEC/EST

```
t = float(60*60*24)
x = p.x_loca + (p.x_vel * t)
y = p.y_loca + (p.y_vel * t)
z = p.z_loca + (p.z_vel * t)
p.set_location(x, y, z)
```

Data structures and Files

I have used lists as the main data structures for storing all of the planet objects in the world. The Planet object itself stores all of the necessary data and attributes of the planet, and this is easily called by parsing through the list and selecting the wanted element and then using the wanted element on this planet object. Eg. calculating the force exerted on a planet “p” by all of the other planets in the list of planet objects presented above in the calc_force() algorithm.

I also created my own data structure in the format of a text file. This was to allow the user to store information about all of the heavenly bodies in a world and easily access the data on the go. The save file structure is explained in more detail in the Savefile_structure_plan.txt file, but in conclusion the structure is similar to the one we used in the course assignments on round 6 (human writeable). Each planet or heavenly body is represented and flagged as being between the “PLN” and “PLEN” headers. Within these headers all of the necessary data of the object is presented in a format that the LoadSaveIO() class can easily read through and finally bring the objects into the simulation. The save file also contains information on the version of the program, as well as the save date of the save file and some information of the world.

Included in the Savedata folder are 4 ready made INIT files which can all be used for testing the program. All of these files have the same structure. The save file structure is explained in more detail in the Savefile_structure_plan.txt file, and with this structure the user may create save files on their own. Using a file like this to load the scenario from, however is required for the simulation to work.

Testing

In the beginning I developed the save file structure side by side with the LoadSaveIO reader. These parts of the program as well as the Planet and World classes and methods were tested simultaneously through a sort of main file. As soon as the reader read through the data of the save file and loaded it into the World like I wanted it to, I continued on to the development of the UI.

Since the actual physics of the simulation were easiest to test with a working graphical interface I next put all effort into making this. To test it I drawed basic shapes on the scene at first and later developed the PlanetGraphisItem for drawing purposes. Now the only part left was to start implementing a run function with a Timer that makes the simulation run smoothly. Once I got this part working I could actually start testing the physics of the sim and the algorithms. In the beginning it was apparent that the physics were botched since all of the planets kept drifting off into oblivion. However, with extensive debugging with the pycharm debugger and loads of printlines I got the physics and the algorithms to work and could actually test the simulation part of only the planets.

Once I had the simulations running according to what I wanted the only thing left to implement was loading a satellite into the program. This was done in a similar way as how I added other graphics items into the scene and thus would be updated (drawed) onto the scene simultaneously with the rest of the planet objects. I was finally satisfied when I tested adding a satellite with similar values as mercury and I could see that the satellite actually orbited the sun in a similar way as mercury does. This meant that the physics also worked for the satellite and now the user may launch a satellite from any given point in the solar system with any given speed in any given direction, and directly see the results of this! (at least in the x,y plane)

The process of testing the program followed the initial plan quite well, with holes only being in the part where I had to implement and test the save file and reader functionalities. Although, the program doesn't quite restrict the user inputs as I would've wanted it to.

Known shortcomings and flaws

The most significant shortcoming in the program has to be the collision detection system. Since I couldn't simply have a graphical collision detection, because all of the bodies interact in the 3D plain in the backend of the program, I had a very hard time getting such a system to work. I tested and tried debugging a couple of ideas for the system, but in the end this was what I worked on last in the project and since the deadline was approaching I didn't have further time to get this up and running. The initial things I tested caused the program to crash in ways it had never done before, so I ended up scrapping all of this to make the code cleaner, and allow the user to still spawn in satellites.

My plan for this was to implement the collision_detection_system as a method of the World class. This method would be called in the update_all() method after the program has finally calculated and changed the location of each object in the world.

It would then check if any of the objects had coordinates so close to each other, that either one of the objects radiuses would overlap the others it would raise an error or terminate the program or something similar. The problems I had with this so far was that I couldn't call this method separately for each planet since some planets would then have their old locations still. The comparison of the planet's radiuses and coordinates also ultimately caused a headache, and I think this was the part that was causing the most problems. With more time I'm sure I could've got this part working!

Some other minor flaws of the program that I know of are the input checking. As of now the program doesn't really restrict the user from inputting crazy values. For example when picking the simulation parameters a user could be careless and add negative values or zero and cause the program to crash. The same also applies to the satellite parameters. I would've wanted to (and still might in the future) have the program check the user inputs to be logical and if not maybe raise an error popup window prompting the user to input some new values. Also I would've wanted to create a more responsive save file reader, to make debugging of the users own save files a bit easier for the user. However, these minor flaws are all finishing touches to the program that I will probably work on in the future as a hobby, as well as creating an installation guide for eg. family or friends so they can also try out the program. Or even making this into a web-based application. For now I will have to assume that both I and the assistant testing the code will understand to not input crazy values in the parameters and upon creating a scenario of their own, will follow the structure of the save file precisely.

3 best areas

- My perhaps favorite aspect of the program is how user friendly it ultimately is. The amount of control the user has of the simulation is sufficient and the possibility to upload custom scenarios based on the imagination of the user gives the user a large range of freedom. It hopefully also encourages the user to study the properties of the scenario from different angles.
- I also put quite some time into making the UI actually look good, and placing all of the fields and buttons neatly and having the inputs be accepted by button clicks. I even added a small feature to ask the user if they are sure they want to exit the program, upon clicking the big red x.
- The graphical scaling of the distances between planets is accurate and was relatively easy to make functiona. I had to scale down the biggest planets radiuses (jupiter, saturn, sun) in order for them to not obscure the whole window, but this just makes the simulation easier to view.

Benjamin Åkerlund
665762 ELEC/EST

3 worst areas

- I don’t quite like the fact that the planet's sizes (radiuses) do not correspond to the scaling of the distances between planets. However, this had to be done in this way for now in order for the simulation to actually be viewable, and since this scaling is only done in the frontend of the program, it doesn’t impact the actual physics of the simulation.
- I also don’t like the scroll bars that the window sets on my graphics view. I struggled with this for a good while, until I finally decided It wasn’t as meaningful as other things I needed to work on. For now everything is at least visible when the user loads a scenario.
- I would’ve wanted to add the opportunity for the user to change the scaling of the graphics view in realtime. This would’ve been quite cool to have with if the user eg. wanted to track a satellite near the outermost planet in the system. The idea for future development is to let the user eg. click on a planet (from a list in the input section) and then the program would display that planet in the middle of the window

Changes to the original plan

Developing this project for the most part went quite according to plan. I didn’t quite have time to implement as much into the program as I would’ve wanted to, but what I did accomplish still looks nice and works fine. My assessment of time taken for this development was largely wrong and also the courses planned ~78hrs of time for the project was inaccurate. What I have so far took around 100hrs to produce. Although, a large portion of this time was spent learning how to use the PyQt5 library since the UI is so critical for the project. I would’ve liked to have more education in PyQt5 along the course and course assignments, as I feel what learned was somewhat lacking in that area,

Realized order and scheduled

The time span of working on this project was largely shifted to the fifth period for my part. The time available for me in the third and fourth period simply wasn’t enough to keep working on this final project. I would say most time spent on the project was in the second half of April and beginning of May. In May I was more comfortable to start pushing the progress so far to Git, since this is the time when everything came together (mostly after debugging).

Benjamin Åkerlund
665762 ELEC/EST

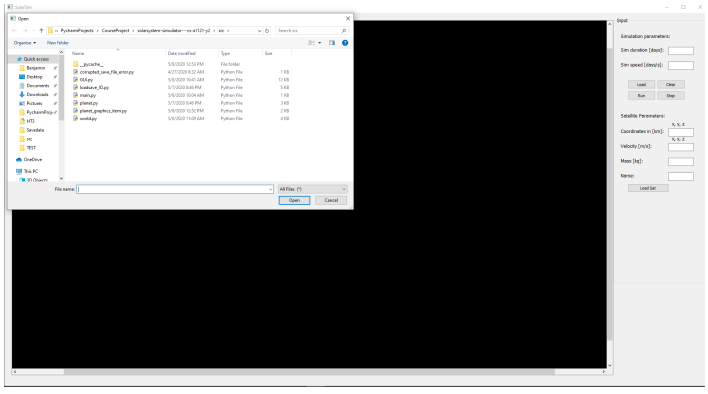
Final assessment and summary

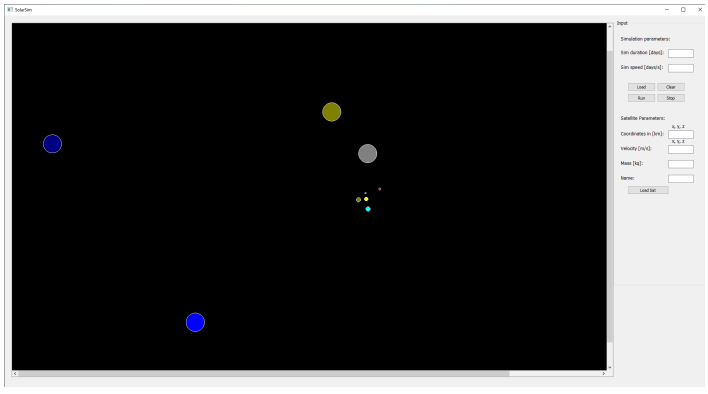
All in all the project turned out quite nice. I am satisfied with how it looks and how it works. I feel that I implemented a lot of “extra” features and things that may not have been necessary for the given assignment, but were extremely necessary for my own satisfaction with the program. In light of the program's shortcomings and flaws, the base concept still works and looks nice. The UI is easy to use and offers the user sufficient control of the simulation. The structure of the program is still suitable for making changes eg. more accurate algorithms for calculating the planetary locations, or also adding more color variants corresponding to the QColor() library.

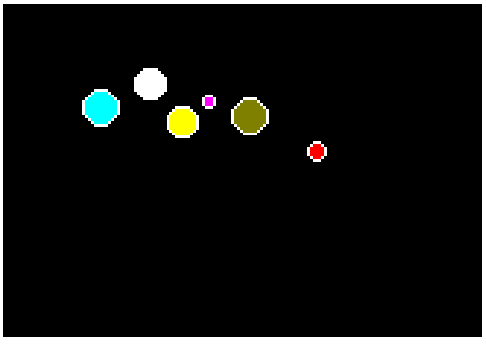
The experience has been extremely educational for me, as I feel I have learned far more during this phase of the course, than any other phase. I set out with a goal of making a solar system simulator with low expectations of it even looking relatively good (comparison with a gray window and dots on it), and ended up with a final product that I can proudly present to my friends and family and say “Look what I made!”. I am also still greatly inspired by this project and will most definitely continue the development of this program, as a hobby. Who knows, maybe I will even someday upload the program online and get feedback from the large community of python developers.

Attachments

Below are screen captures of loading a file into the simulator, the initial scenario, adding a satellite (white) and the scenario after letting the simulator run for a couple of years

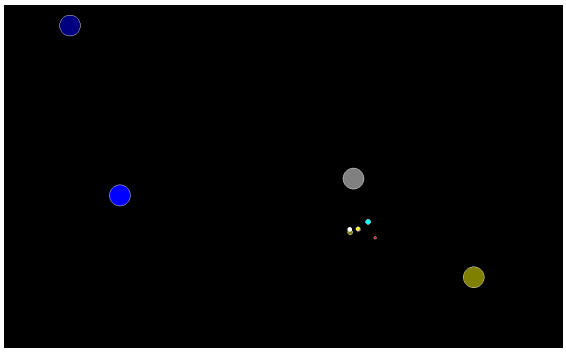


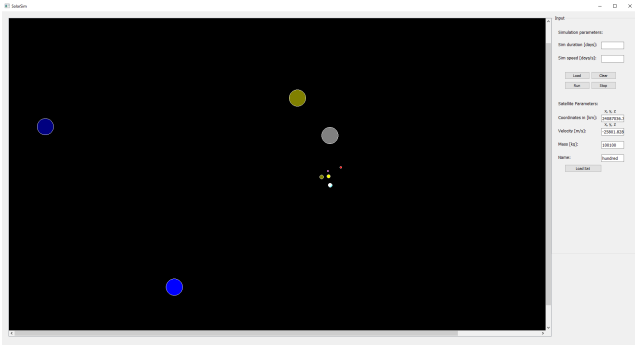




Screenshot of a satellite loaded into the simulation (white) next to the earth and sun (cyan and yellow)

Benjamin Åkerlund
665762 ELEC/EST





References:

https://en.wikipedia.org/wiki/Gravity

https://en.wikipedia.org/wiki/Newton%27s_laws_of_motion

https://en.wikipedia.org/wiki/Velocity

http://www.wikicalculator.com/formula_calculator/Euclidean-distance-between-two-points-in-three-dimensional-(3D)-space--with-Cartesian-coordinates-221.htm

https://theskylive.com/3dsolarsystem

https://www.solarsystemscope.com/

https://space.jpl.nasa.gov/

http://gafferongames.com/game-physics/integration-basics/

https://github.com/lukekulik/solar-system

https://gist.github.com/CodeDotJS/64f0d3d86d05b93af3b6

https://www.101computing.net/solar-system/

https://www.astropy.org/

https://doc.qt.io/

https://www.theplanetstoday.com/planets_information_basic_facts.html