Orbits in Curved Space Time

Aleksas Girenas

December 16, 2019

1 Abstract

In 1915, Albert Einstein's General Theory of Relativity extended Newtonian gravitation theory, revealing previously unanticipated subtleties of nature. Einstein's theory explained the perihelion advance of Mercury and the tiny discrepancy in the orbit of Mercury was actually the first evidence for what lay beyond Newtonian gravitation. Shortly after Einstein published his equations a solution was found by Schwarzchild for the gravitational effects by a spherical mass. I attempted to simulate the effects on a test particle when orbiting a spherical mass in python using PyQt5 [4] as the GUI.

2 Introduction

To develop a program that visualises orbits around curved space time we must first understand the fundamental mathematics used. The Schwarzchild geometry is able to describe orbits in curved space time and we can form geodesics to describe the movement of a test particle [1]. A geodesic is the path that a particle will follow. We can describe the geometry by the metric (in units where the speed of light is one, $c = 1$):

$$ds^2 = -(1 - r_s/r)dt^2 + \frac{dr^2}{(1 - r_s/r)} + r^2 do^2 \tag{1}$$

The quantity ds denotes the invariant spacetime interval, an absolute measure of the distance between two events in space and time, t is a 'universal' time coordinate, r is the circumferential radius defined so that the circumference of a sphere at radius r is $2\pi r$. We must however create our Attractor and therefore for a mass M (for a static, spherically symmetric mass distribution)[2]:

$$ds^2 = -(1 - 2M/r)dt^2 + \frac{dr^2}{(1 - 2M/r)} + r^2(do^2 + \sin^2(\theta)\phi^2) \tag{2}$$

We must also understand the equation for forming a geodesic so that we know the trajectory of the particle[2]:

$$\frac{d^2x^\mu}{d\lambda^2} + \Gamma^\mu_{\alpha\beta}\frac{dx^\alpha}{d}\frac{dx^\beta}{d\lambda} = 0 \tag{3}$$

for some affine parameter λ (that removes certain terms that would otherwise appear). The symbol $\Gamma^\mu_{\alpha\beta}$ is made from a product of the metric tensor and its derivatives. The metric tensor describes the gravitational field and $\Gamma^\mu_{\alpha\beta}$ is the generalisation of the Newtonian gravitational force components. This allows us to form a geodesic to find the motion of the particle. These equations form the basis for calculating the trajectory of the test particle.

3 The Application

To make the application easier to understand and develop the Mathematics of plotting the particle was split from the GUI code. Therefore we are left with two main files: **main.py** which launches and runs the flow of the application and includes the GUI; **Plotting.py** which includes the **EinsteinPy** packages and runs the calculations for the plots. This also makes the code very easy to read and modify. The following functions are also much clearer and better documented in the code.

3.1 GUI

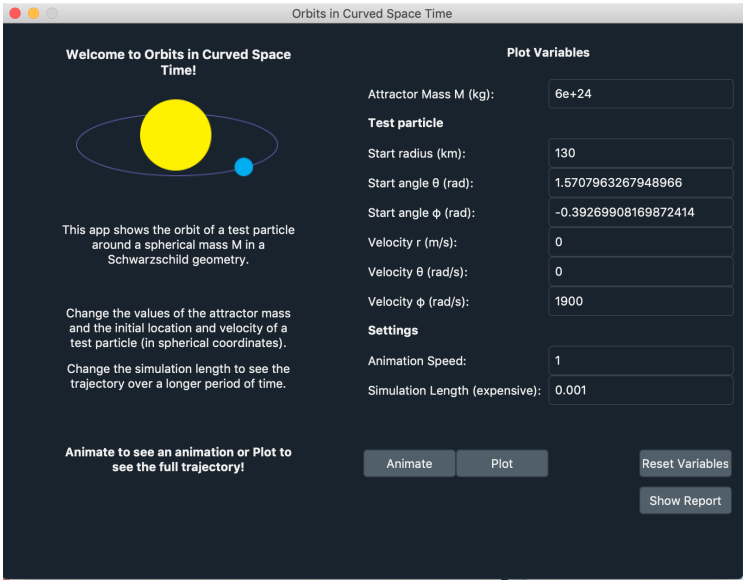

Figure 1: The main window

A UI is designed to make it easier for a user to interact with the program. Therefore I took care to make the application as fun and intuitive to use to show results faster and understand how the change of inputs affect the plots. The use of stylesheets and images makes the UI clearer and prettier.

By importing the **Plotting.py** file into the **main.py** script we can access and change it's values, including running the functions for plotting the orbits; however, we must first create an instance of the plotting file so that the variables are stored in memory: if they are changed, calling the functions in our instance will use the same variables.

As python is a procedural language and PyQt5 runs in an object orientated approach the **main.py** file first runs the MainWindow() class (child of QMainWindow) and initialises the UI. The main() loop is then run which starts the application.

To allow for fine tuning of the UI I found it easier to hard code the positions of buttons and inputs and the setup process of the UI. The buttons call the plotting functions in **Plotting.py** after setting all the values of the plotting file to our text inputs. When

using the program you can choose to either get static plots that are plotted over a period of time set by the Simulation Length or an animation of the trajectory which will show you the velocity of the orbit. We can reset the values to their defaults by pressing the Reset Variables button which simply changes the text fields to the hard coded default values. This does not change the variables for plotting as the plotting buttons call the function to do that and it would be unnecessary to call it twice.

```python
def ButtonClicked(i):
    QApplication.processEvents()
    '''Write values from input. i = 1 -> plot; i = 2 -> animate; else -> open report.
       Functions in Plotting are run on a separate thread to prevent main thread from stalling'''
    SetTextValues()

    if i == 1:
        callback.RunPlot()
    elif i == 2:
        callback.RunSimulation()
    elif i == 3:
        #Reset values to default
        callback.attractorMass = 6e24
        callback.radius = 130
        callback.theta = np.pi/2
        callback.phi = -np.pi/8
        callback.v_Radius = 0
        callback.v_Theta = 0
        callback.v_Phi = 1900
        callback.animInterval = 1
        callback.sLength = 0.001
        self.line2.setText(str(callback.attractorMass))
        self.line3.setText(str(callback.radius))
        self.line4.setText(str(callback.theta))
        self.line5.setText(str(callback.phi))
        self.line6.setText(str(callback.v_Radius))
        self.line7.setText(str(callback.v_Theta))
        self.line8.setText(str(callback.v_Phi))
        self.line9.setText(str(callback.animInterval))
        self.line10.setText(str(callback.sLength))
    else:
        webbrowser.open_new(file)
```

Figure 2: Function for when a button is clicked

3.2 Plotting the orbits

I used the library **EinstenPy** [3] that contains the necessary equations discussed earlier in eq.(1)(2)(3) to form an attractor of mass M, a test body at a given position and velocity, and the Schwarzchild geometry to make the application cleaner and easier to write and modify.

The Body() function in Attractorsetup() creates a generic body of mass M that will be used as the spherical mass that acts on the body as described by eq.(2):

```python
def AttractorSetup(massA):
        '''Creates attractor with given mass'''
        Attractor = Body(name="Attractor", mass=massA * u.kg, parent=None)
        return Attractor
```

Figure 3: Creating an attractor

The Schwarzchild geometry is most easily represented in a spherical coordinate system and therefore it is easier to create the particle in the same coordinate system. The SphericalDifferential() in the BodySetup() function is used to calculate and transform the velocity of a test particle in spherical coordinates. By taking in inputs of the starting position (radius, angle θ, and angle ϕ) and velocity (along the: radius; angle θ; and angle ϕ) the values are stored as a body.

```python
def BodySetup(radius, theta, phi, v_Radius, v_Theta, v_Phi):
        '''Creates test particle with initial position and velocity vectors of test partcle in spherical coordinates
        Prefix v is velocity in respective directions'''
        return SphericalDifferential(radius*u.m, theta*u.rad, phi*u.rad, v_Radius*u.m/u.s, v_Theta*u.rad/u.s, v_Phi*u.rad/u.s)
```

Figure 4: Creating the test particle

Both the BodySetup() and AttractorSetup() are then passed into the plotting functions that creates a new body attaching the differential of the test particle to the attractor. This new body is then used to create the geodesic (3) that follows the geometry (1): following Schwarzchilds equation there is a "sink" in the geometry where the Attractor of mass M is and the geodesic is the path that the particle will follow. By plotting the function over time with a set step size we can see the trajectory. The animation function is analogous to the plotting function except for being animated which takes an interval between frames and can be edited in the GUI by changing the 'animation speed'.

```python
def Plot(Attractor, sph_obj, simLength = 0.002):
        '''Plots with given attractor and bodies. Note: increasing simLength gives longer simulation but takes longer to load'''
        plt.close()
        obj = StaticGeodesicPlotter()
        #Adding the body to the attractor
        Object = Body(differential=sph_obj, parent=Attractor)
        #Define geometry i.e. Schwarzschild
        geodesic = Geodesic(body=Object, time=startTime * u.s, end_lambda=simLength, step_size=sSize)
        #Plotting the trajectory
        obj.plot(geodesic)
        plt.title("Plot of Orbit")
        #Plot Visuals
        fig = plt.gcf()
        fig.canvas.set_window_title("Plot of Orbit")
        plt.show()
```

Figure 5: The final plotting function

3.3 Understanding the plots

The plots show intuitively the orbit of a test particle in curved space time. According to the Schwarzschild metric, the proper radial distance, the actual distance measured by an observer at rest at radius r between two spheres separated by an interval dr of circumferential radius r is $dr/\sqrt{1-r_s/r}$. This is larger than the radial interval dr expected in a flat, Euclidean geometry making the geometry 'stretched' in the radial direction as observed in the orbit of the particle.

Certain phenomenon to observe in the results are: with a smaller mass we can observe generic elliptical orbits seen with Newtonian gravity fig.6; with larger masses we can observe the precession of the orbit as the apastron (greatest distance from the attractor) moves - similar to what can be seen with the orbit of Mercury fig.7; we can observe the 'slowing' of the particle when it gets close to the attractor of high mass; a non rotating black hole can be simulated and a particle can be lost as it approaches the event horizon fig.8 (unfortunately we cannot simulate what happens beyond)!

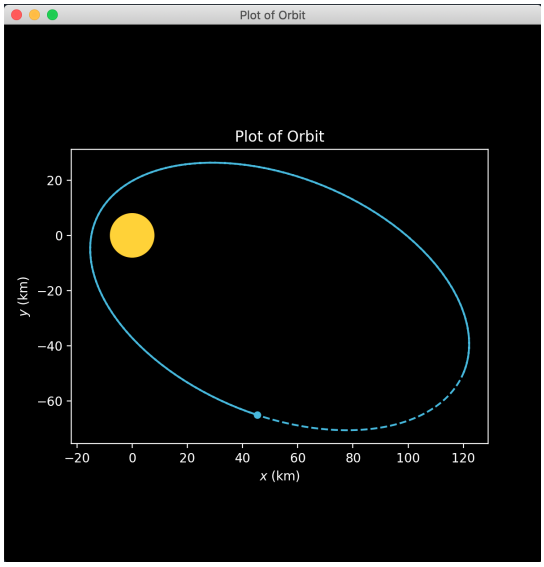

Figure 6: A more generic elliptical orbit

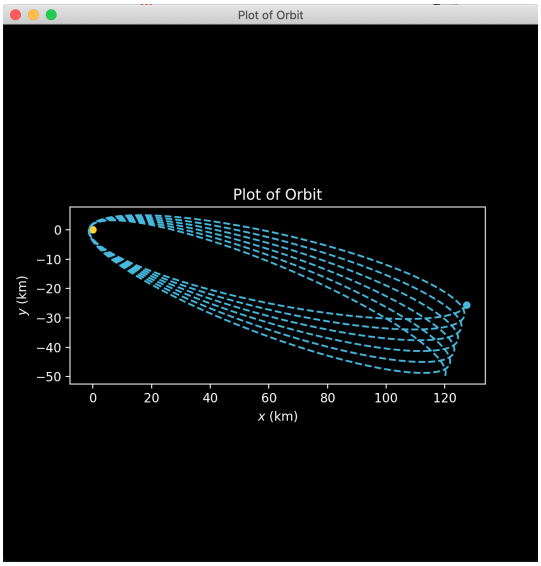

Figure 7: Precession of orbit observed

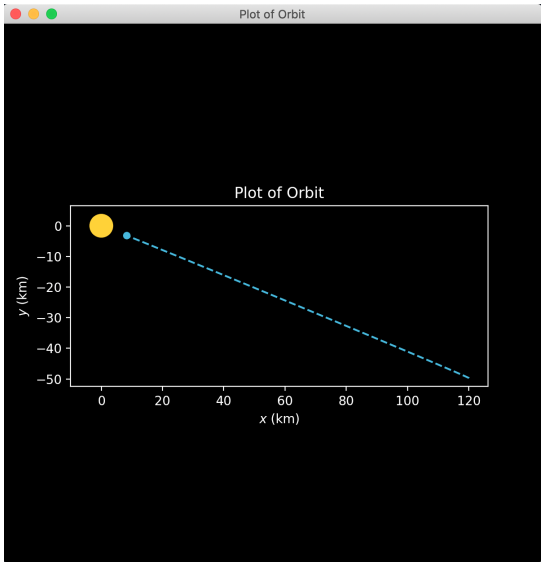

Figure 8: Approaching a large mass

4 Limitations

I was not able to insert the plotting windows into the GUI as in **EinsteinPy** the StaticGeodesicPlotter() class is not a child of **Matplotlib** plotting classes; instead it uses the plotting functions from **Matplotlib** which means that a new figure is created (i.e. a new window) when the **EinsteinPy** plotting and animation functions are called. Therefore I chose to make the application work well and look good with separate windows: the plot windows are simple and clearly show the plot; when a new plot is shown the old one is closed; the main GUI window is easy to use and intuitive.

Attempts to prevent the main window from freezing when a plot was being calculated included moving the plotting calculations onto a separate thread. This did not work as by default **Matplotlib** which is also the basis for the **EinsteinPy** plotting must be run on the main thread for safety reasons [5]. This means that the GUI freezes when plotting calculations are being done until the plot is complete. In order to fix this the backend for rendering in **Matplotlib** must be edited to allow for it to run on a different thread. Due to time limitations this addition was not done.

Intermittently pausing calculations to update the GUI was possible but creating a plot would take much longer as a result.

5 Conclusion

I was able to produce a program to show the orbit of a test particle in curved space time around an object of high mass. This application is useful to simulate and visualise the multiple effects of General Relativity that can be observed in the real world in a simple and easy to use package. The design of the code allows for easy modifications to improve the simulation and add to/change the GUI. In order to achieve best results however, it may be easier and better to use an entirely different framework of libraries and use a language such as C++, as python becomes somewhat difficult to scale for larger applications, where memory management and defined variables may make it easier to write. Perhaps even using a graphics api such as OpenGL for the mathematics would yield much faster and more incredible results especially when calculating tensors. The application does perform well and at a small scale using python was a lot faster to produce a simple program.

References

[1] More about the Schwarzschild Geometry https://jila.colorado.edu/~ajsh/bh/schwp.html

[2] Computers in Physics 6, 498 (1992); doi: 10.1063/1.168437 https://aip.scitation.org/doi/pdf/10.1063/1.168437

[3] EinsteinPy https://einsteinpy.org/

[4] PyQt https://riverbankcomputing.com/software/pyqt/intro

[5] Matplotlib backend https://matplotlib.org/faq/usage_faq.html#what-is-a-backend