Overview of Some Aspects of BlackHoleOrbits

George Jones

June 29, 2005

1. Solving the Differential Equations

This programme solves the coupled system of differential equations

$$\begin{array}{rcl} \frac{dr}{dt} & = & p \\ \frac{d\phi}{dt} & = & \frac{L}{r^2} \\ \frac{dp}{dt} & = & \frac{L^2}{r^3} - 15\frac{k}{r^2} - 45\frac{L^2}{r^4} \end{array} . \tag{1}$$

Here, r and ϕ give the position of the particle using standard polar coordinates for the plane, L is the relativistic orbital angular momentum of the particle, $k = 1$ for the massive particle case, and $k = 0$ for the case of light. The variable p is used to make all the equations first-order.

To see how to solve this system on a computer, consider an equivalent version of the second equation:

$$\lim_{\Delta t \to 0} \frac{\Delta\phi}{\Delta t} = \frac{L}{r^2}. \tag{2}$$

If Δt is small, then

$$\frac{\Delta\phi}{\Delta t} = \frac{L}{r^2} \tag{3}$$

is a good approximation. Writing $\Delta\phi = \phi_2 - \phi_1$ and rearranging gives

$$\phi_2 = \phi_1 + \frac{L}{r^2}\Delta t. \tag{4}$$

Applying similar considerations to each equation in the system results in

$$\begin{array}{rcl} r_2 & = & r_1 + p\Delta t \\ \phi_2 & = & \phi_1 + \frac{L}{r^2}\Delta t \\ p_2 & = & p_1 + \left(\frac{L^2}{r^3} - 15\frac{k}{r^2} - 45\frac{L^2}{r^4}\right)\Delta t \end{array} . \tag{5}$$

Solving the exact system (1) numerically on a computer involves repeatedly iterating the approximate system (5). A first attempt[1] might be:

[1] This, and other programming aspects, are written in Java-like pseudocode, **not** Java.

```
while(appropriate condition){
    r = r + p*dt
    phi = phi + L/r/r*dt
    p = p +(L*L/r/r/r - 15*k/r/r - 45*L*L/r/r/r/r)*dt
}
```

This first attempt doesn't quite work because r is updated before r is used in the equations that update ϕ and p, while the pre-updated value of r is needed to update ϕ and p. Also, initial values for r, p, and ϕ are needed to start things off.

```
phi = PI/2 // starts the particle off directly above black hole
input r, v0, launch angle; //implement a gui to do this
calculate_p()
while(appropriate condition){
    phi = phi + L/r/r*dt
    p = p +(L*L/r/r/r - 15*k/r/r - 45*L*L/r/r/r/r)*dt
    r = r + p*dt
}
```

This is Euler's method for solving differential equations, which was the method I first used to get the programme up and running, and which, for simplicity, I use below when other aspects of the programme are illustrated. After the programme was working, I implemented the fourth-order Runge-Kutta method for solving differential equations in order to achieve better accuracy and speed.

2. Displaying the Result

The animation display region should be thought of in 2 complementary ways: as a rectangular portion of the computer screen that consists of pixels; as a rectangular region of the space around the black hole where distances are measured in kilometres. Each of these ways has its own set of coordinates, and the method for converting between these sets of coordinates follows.

The animation is displayed in a 500pixel $\times$ 500pixel region of the computer screen. The computer uses x versus y Cartesian screen coordinates for this region, but in a slightly non-standard (for mathematicians, maybe not for computer scientists) way. The origin for the screen coordinates (x, y) is at the top left of the display, x increases horizontally from left to right, and y increases vertically from top to bottom. Consequently, the screen coordinates of the display's: top left corner are $(0, 0)$; top right corner are $(499, 0)$; bottom left corner are $(0, 499)$; and bottom right corner are $(499, 499)$.

The rectangular region in space is $620\,\text{km} \times 620\,\text{km}$ and has standard Cartesian coordinates whose origin resides in the middle of this region. Thus, the physical coordinates $\left(x_{\text{physical}}, y_{\text{physical}}\right)$ of the display's: top left corner are $(-310, 310)$; top right corner are $(310, 310)$; bottom left corner are $(-310, -310)$; and bottom right corner are $(-310, 310)$.

Comparing screen and physical widths of the display region gives the unit conversion between physical distances in kilometres and screen distances in pixels:

$$\begin{array}{rcl} 620\,\text{km} & = & 500\text{pixels} \\ 1\,\text{km} & = & \frac{500}{620}\text{pixels} \end{array} . \tag{6}$$

It might seem reasonable to put

$$\begin{array}{rcl} x & = & \frac{500}{620}x_{\text{physical}} \\ y & = & \frac{500}{620}y_{\text{physical}} \end{array} , \tag{7}$$

but this doesn't take into account that the physical origin has screen coordinates $(x, y) = (250, 250)$, nor does it take into account that the screen coordinate y increases in the down direction, while the physical coordinate y_{physical} increases in the up direction.

$$\begin{array}{rcl} x & = & \frac{500}{620}x_{\text{physical}} + 250 \\ y & = & -\frac{500}{620}y_{\text{physical}} + 250 \end{array} . \tag{8}$$

The 250's give the offset between the physical origin and the screen origin, and the negative sign gives changes the sense of the y direction. Note that since a pixel has non-zero width, the physical coordinates of the corners of the display region do not quite map into the screen coordinates of the corners.

The relationship between the Cartesian coordinates of physical space and the polar coordinates used in the system of differential equations above is $x_{\text{physical}} = r\cos\phi$, $y_{\text{physical}} = r\sin\phi$. These are used to plot on the screen the numerical solution to the system of differential equations once every iteration of the `while` loop.

First an explanation of what happens when the trail is turned on. Every iteration of the `while` loop, a yellow pixel is turned on at the screen location of the orbiting particle, and, if `dt` is small enough, several iterations of the `while` loop are needed to produce a change in x (or y) larger than the width of 1 pixel. Therefore, the same pixel is turned on for several iterations of the `while` loop before an adjacent pixel is turned on. The result is a continuous trail.

```
while(appropriate condition){
    put_yellow_pixel(r*cos(phi)*500/620 + 250 , 250 - r*sin(phi)*500/620)
    r = r + p*dt
    phi = phi + L/r/r*dt
    p = p +(L*L/r/r/r - 15*k/r/r - 45*L*L/r/r/r/r)*dt
}
```

When the trail is turned off, every iteration of the `while` loop, a small white circle is turned on at the screen location of the orbiting particle, and a small black circle is "turned on" at the previous location of the orbiting particle. If a small black circle were not "turned on" at the previous location of the orbiting particle, then a thick white particle trail would appear. If `dt` is small enough, the combination of white and black circles make the white circle appear to move smoothly through space.

```
while(appropriate condition){
    put_black_circle(old_r*cos(old_phi)*500/620 + 250 , 250 -
                old_r*sin(old_phi)*500/620)
    put_white_circle(r*cos(phi)*500/620 + 250 , 250 - r*sin(phi)*500/620)
    old_r = r
    old_phi = phi
    r = r + p*dt
    phi = phi + L/r/r*dt
    p = p +(L*L/r/r/r - 15*k/r/r - 45*L*L/r/r/r/r)*dt
}
```

This concludes the overview. The devil is in the details.