Simulating Kerr Geodesics

Ian Smith

June 2015

Abstract

A simple and accurate numerical method is presented aimed at generating 4D geodesics for particles and photons in the Kerr spacetime. Particular attention is paid to the problem of generating a variety of usable initial conditions for the simulator. Sources of error are quantified and monitored as a matter of course. The source code for the suite of programs is publically available under a BSD licence.

1 Bound Orbits in the Kerr Spacetime

The starting point is the Kerr line element expressed in terms of the commonly used Boyer-Lindquist coordinates.

$$\begin{aligned} ds^2 = \frac{\sin^2\theta}{r^2 + a^2\cos^2\theta}(adt - (r^2 + a^2)d\phi)^2 + \frac{r^2 + a^2\cos^2\theta}{(r^2 - 2Mr + a^2)}dr^2 \\ + (r^2 + a^2\cos^2\theta)d\theta^2 + \frac{(r^2 - 2Mr + a^2)}{r^2 + a^2\cos^2\theta}(dt - a\sin^2\theta d\phi)^2 \end{aligned}$$

The geodesic equations for the Kerr spacetime are complex second order differential equations which are well known and can be solved using standard numerical integration techniques, typically one of the Runge-Kutta methods. Unfortunately this approach is unsuitable for studying horizon crossing scenarios because in these cases the t variable becomes "unresponsive" near the outer horizon and slows the simulation to a near standstill without ever getting there.

Rather than attempting to formulate and solve these geodesic equations directly, the approach taken here is to evolve the first order equations of motion expressed in terms of the constants of motion E (energy), L (equatorial angular momentum), and Q (Carter's constant). Because t is not an integration variable here, these equations do not grind to a halt at the horizon.

The approach is so straightforward that the complete simulator script consists of just over a hundred lines of Python code [6].

1.1 Equations of motion

This is the well known set of equations describing Kerr orbits in terms of E, L, and Q:

$$\frac{dt}{d\lambda} = \frac{(r^2+a^2)\left((r^2+a^2)E - aL\right)}{(r^2-2Mr+a^2)} + a(L - aE\sin^2\theta) \tag{1}$$

$$\frac{dr}{d\lambda} = \pm\sqrt{R(r)} \tag{2}$$

$$\frac{d\theta}{d\lambda} = \pm\sqrt{\Theta(\theta)} \tag{3}$$

$$\frac{d\phi}{d\lambda} = \frac{a\left((r^2+a^2)E - aL\right)}{(r^2-2Mr+a^2)} + \left(\frac{L}{\sin^2\theta} - aE\right) \tag{4}$$

where

$$d\lambda = \frac{d\tau}{(r^2+a^2\cos^2\theta)} \tag{5}$$

These are essentially equations 2 and 3 from [1], fully expanded except for the two potential functions R and Θ which, together with their differentials, are defined in equations 6-9 below.

Notice also the use of the "Carter-Mino" time variable, λ, to render the R and Θ equations (plus their differentials of course) mutually independent. In terms of this time variable, the potentials are simply the squares of the r and θ velocities.

$$R(r) = \left((r^2+a^2)E - aL\right)^2 - (r^2-2Mr+a^2)\left(Q + (L-aE)^2 + \mu^2 r^2\right) \tag{6}$$

$$\frac{dR(r)}{dr} = 4rE\left((r^2+a^2)E - aL\right) - (2r-2M)\left(Q + (L-aE)^2 + \mu^2 r^2\right) - 2\mu^2 r(r^2-2Mr+a^2) \tag{7}$$

$$\Theta(\theta) = Q - \cos^2\theta\left(\frac{L^2}{\sin^2\theta} + a^2(\mu^2 - E^2)\right) \tag{8}$$

$$\frac{d\Theta(\theta)}{d\theta} = 2\cos\theta\sin\theta\left(\frac{L^2}{\sin^2\theta} + a^2(\mu^2 - E^2)\right) + \frac{2L^2\cos^3\theta}{\sin^3\theta} \tag{9}$$

The t and ϕ equations (1 and 4) can be evolved trivially using the Euler method, but the R and Θ equations (2 and 3) require a little more effort. Because of the square roots the Euler approach is severely hampered by the need to identify turning points, which is difficult to achieve reliably, and even harder to do without compromising the accuracy of the simulation. The approach described here is to use a symplectic Stormer-Verlet integrator [2] to evolve the R and Θ equations, which turns out to be a surprisingly simple solution, and is possible because the potential equations (when squared) are both of the form:

$$\dot{x}^2 - V(x) = 0 \tag{10}$$

This expresison is used to quantify the integration errors, whilst the two differentiated potentials in equations 7 and 9 are used for velocity updates in the integrator routines.

The simulator uses composition [2] to step the (even) integration order from 2 in the case of basic Stormer-Verlet, up to a maximum of 10th order.

2 Finding initial conditions

Another difficulty in generating orbits is the problem of generating initial conditions. Finding an interesting set by trial and error is hard, partly because many combinations of the constants of motion are unphysical.

Here I present a straightforward way of generating two types of bound orbits in three spatial dimensions. It is based on solving sets of three potential equations under various conditions using the three constants of motion E, L, and Q as variables, and turning points of the potentials as constant parameters.

2.1 Constant radius ("spherical") orbits

For constant radius orbits at r_0, we are looking for a double root (local maximum) of the quartic $R(r)$, in other words we want the radial velocity to be zero, and its differential to also be zero so that the radial speed remains zero during the orbit. $\Theta(\theta)$ will also be zero at the maximum deviation from equatorial (the minimum value of θ).

$$R(r_0, E, L, Q) = 0 \tag{11}$$

$$\frac{dR(r_0, E, L, Q)}{dr} = 0 \tag{12}$$

$$\Theta(\theta_{MIN}, E, L, Q) = 0 \tag{13}$$

2.2 Variable radius ("spherical shell") orbits

For variable radius orbits we are looking for two distinct roots of the quartic $R(r)$, in other words that the orbit is bound between two r values, r_1 and r_2. The $\Theta(\theta)$ condition is unchanged from the constant radius case above.

$$R(r_1, E, L, Q) = 0 \tag{14}$$
$$R(r_2, E, L, Q) = 0 \tag{15}$$
$$\Theta(\theta_{MIN}, E, L, Q) = 0 \tag{16}$$

2.3 Plummeting (an aside)

These ”orbits” are not the focus of this article, but are easily specified by setting $L = 0$ after generating a spherical orbit as in equations 11-13 above.

2.4 Finding the roots

Equations 11-13 and 14-16 can of course be solved by various root-finding techniques, but the equations and algorithms can often become messy and potentially error-prone, with no easy way to check intermediate results. For this work I took a simpler approach; forming a sum-of-squares error function from the constraints described above, minimizing it, then making sure it is sufficiently close to zero. By experience I have found that the Nelder-Mead algorithm from Scipy converges reliably from zero initial conditions on E, L, and Q, and so far have found no reason to look elsewhere.

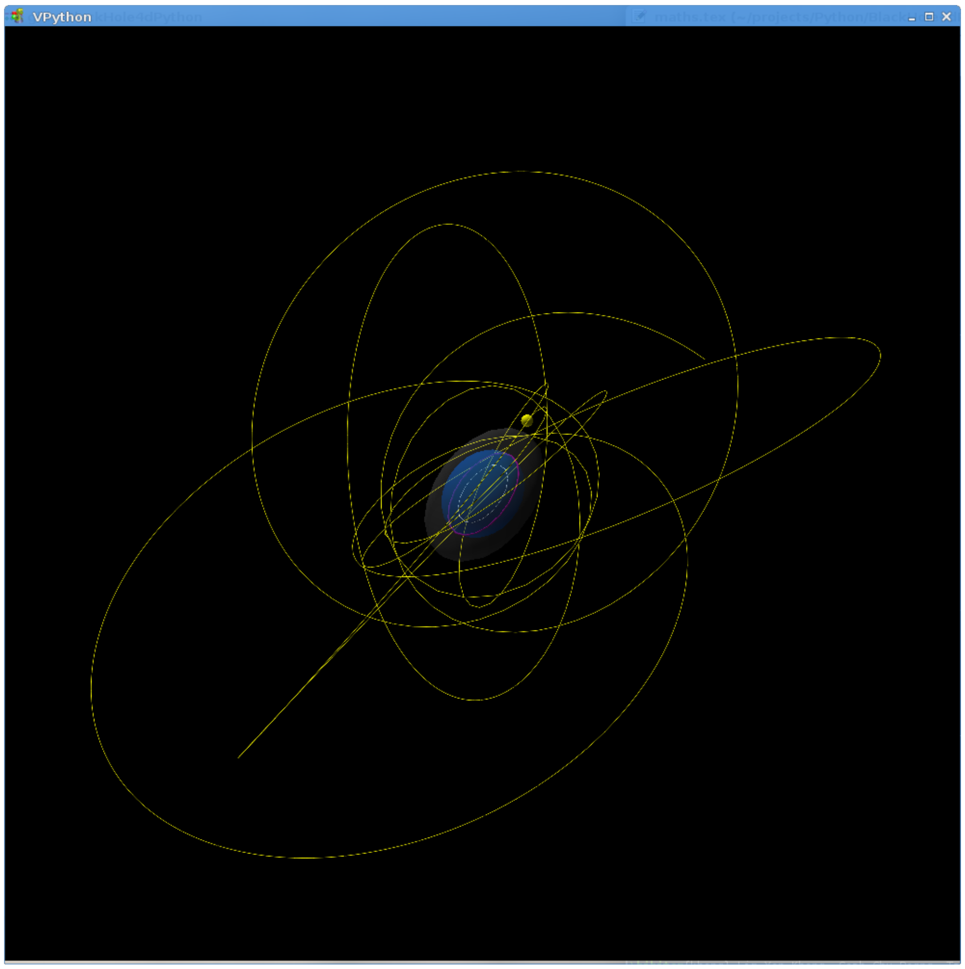
A typical invocation of tthe suite of progrms might be:

```
./genparam.py 3.0 12.0 0.3 1.0 1.0 >test 2>potential; ./rungraphics test
```

which generates an orbit bound between 3 and 12 radial units and with a minimum latitude of 0.3 radians (starting from the ”north pole” of course). Once the optimization has terminated the generation script then writes an initial conditions file (including the final values of E, L, and Q as well as optimizer status) in JSON format for use as input to the simulator. Here is an example:

```
{ "M" : 1.0,
  "a" : 1.0,
  "mu" : 1.0,
  "E" : 0.937868402897,
  "Lz" : 2.16351904692,
  "C" : 2.51243192834,
  "r" : 7.5,
  "theta" : 1.57079632679,
  "time" : 20.0,
  "step" : 0.001,
  "integratorOrder" : 2,
  "error" : 3.106239408e-24,
  "success" : "True",
  "message" : "Optimization terminated successfully."
}
```

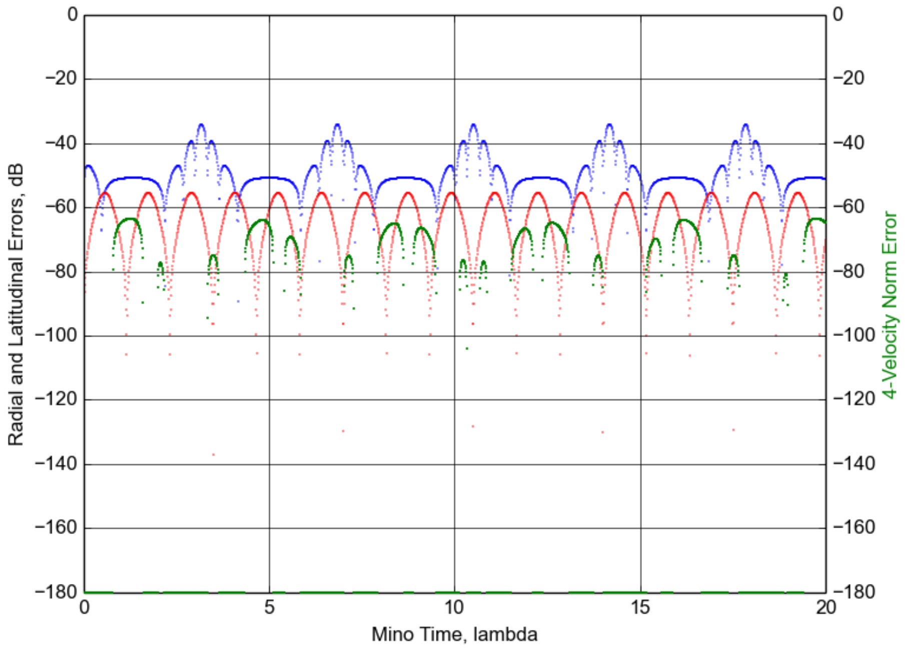
The simulator takes this input, calculates the geodesic and writes its output data in JSON format (for the default timestep the output file is approximately 6.7MB). A static image of the corresponding geodesic is shown below. VPython allows the user to zoom and rotate the view, and watch it evolve. The output of the simulator has been informally



checked against a number of publically available result sets and programs to guard against the possibility of gross errors. This includes but is not limited to the program GRorbits for equatorial geodesics [5], together with published papers for spherical particle orbits [4], and photon orbits [3].

2.5 . . . but are they REALLY geodesics?

Despite the high level of accuracy which the symplectic integrator can achieve for r and θ, this provides no guarantee that the complete solution is an accurate geodesic. To verify this, we can evaluate the four-velocity norm of the path by plugging all the coordinate derivatives into the line element. This error data is part of the JSON output of the simulation. The figure below shows the errors in the simulation: blue is the radial error, red the latitudinal, and green the (deviation from) four-velocity norm, all plotted against a false dB scale (where 0dB = 1.0). This type of error plot makes it easy to see the effects



on accuracy of changing the timestep and integrator order; the default values of 2nd order integration and a fairly course timestep are intended to encourage experimentation.

3 The programs

The sources for the programs [6] described here are publically available on GitHub, under a BSD licence. These are part of a suite of very small scripts and programs which are intended to implement the methods in this article as concisely as possible. There is a short README file in the root directory which gives basic instructions for running the simulations.

References

[1] Wilkins, D.C., ”Bound Geodesics in the Kerr Metric”

[2] Hairer, E., Hairer, M., ”GniCodes - Matlab programs for geometric numerical integration ”

[3] Lim Yen Kheng, Seah Chu Perng, Tan Boon Sze Jackson, ”Massive Particle Orbits Around Kerr Black Holes”

[4] Teo, E, ”Spherical photon orbits around a Kerr black hole”

[5] Tujela, S., ”GRorbits, http://stuleja.org/grorbits/”

[6] Smith, I. C., ”4D Kerr Spacetime Geodesic calculation and display, https://github.com/m4r35n357/BlackHole4DPython.git”