UJTI

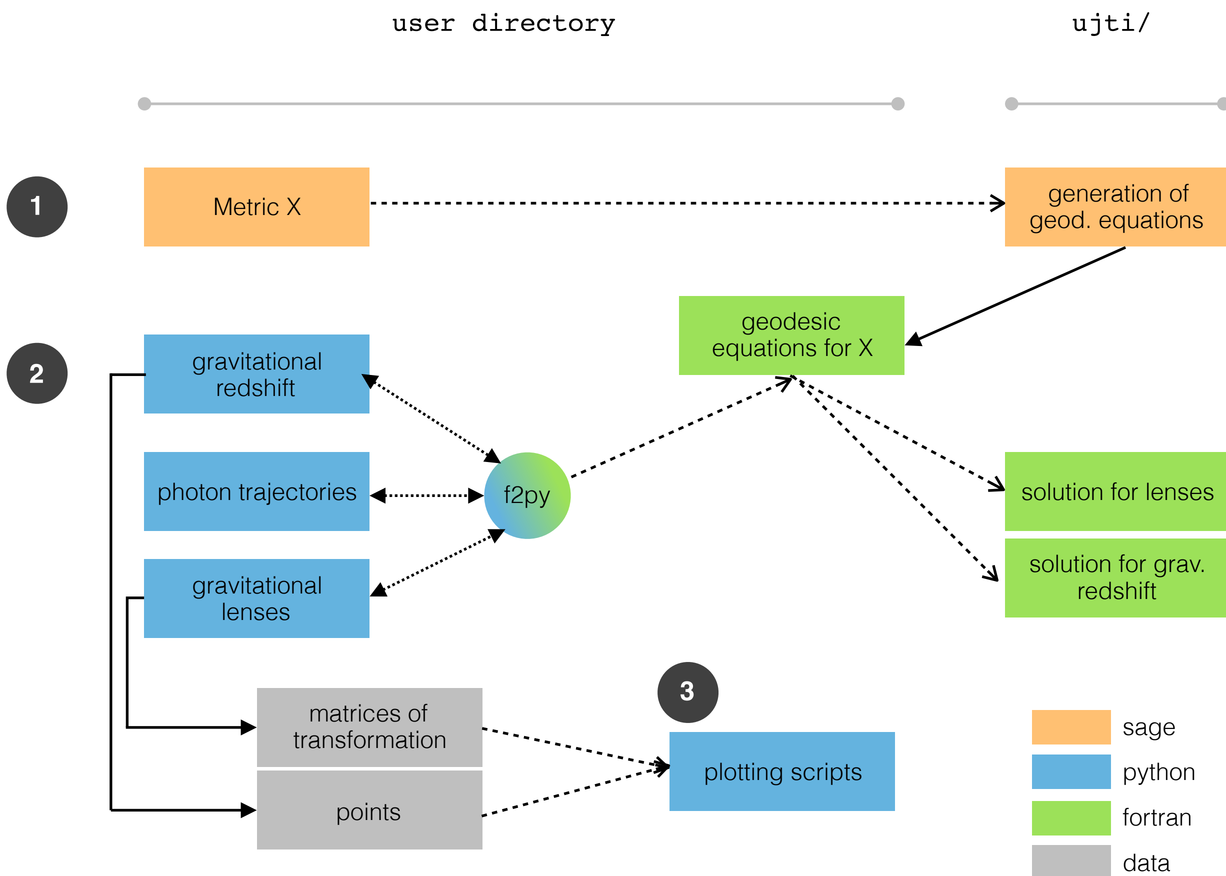
documentation

Last updated: 2017-06-22

Description

Ujti is a library and some scripts that calculate geodesics, gravitational lenses and gravitational redshift in principle, for any metric. Special attention has been given to compact objects, so the current implementation considers only metrics in spherical coordinates.

Structure of the program



1 In Metric X, the Sage script Generation of geodesic equations is called (like a library), which provides classes that specify the metric tensor, and a function that generates the Fortran code Geodesic equations for metric X, which defines the modules and subroutines with the metric and the right-hand side of the geodesic equations.

2. Geodesic equations for metric X also includes the modules that solve for lenses and gravitational redshift for *one* geodesic. Each one of these modules defines functions for a coordinate system of the metric (which must coincide with the one used in Sage), input/output coordinates, and numerical method of solution. Geodesic equations for metric X must be compiled with f2py, which creates a module that can be imported by the Python application scripts, Gravitational redshift, Gravitational lenses and Photon trajectories. In each of these scripts, the initial conditions are varied to later form an image or dataset.

3. The output of each script is then plotted by other scripts, making use of Matplotlib or Gnuplot.

Generating equations for a metric

Warning: the only coordinate system currently implemented in the next steps is spherical coordinates, although the procedure for this section will work in any coordinate system.

The Sage script `geodesic_equation.sage` generates the equations for a metric. There are two components of this script that are relevant for the user:

- `class Spacetime` : defines a Spacetime object, that handles the metric tensor, coordinates and parameters.
- `write_equations()` : calculates the geodesic equations, assembles them and writes the resulting Fortran code to a file.

The complete documentation for both components is transcribed next.

```
Spacetime(coordinates, parameters=[])
Creates a spacetime

        Parameters
        ----------
        coordinates : list of sage symbols
            Symbols of the coordinates to be used. Only the coordinates should be here,
since this list is used to perform derivatives with respect to those variables.
        parameters : list of sage symbols, optional
            Symbols of the parameters of the metric (other variables present in the metric)

        Example
        -------
        sage: x0,x1,x2,x3,M = var('x0,x1,x2,x3,M')
        sage: schwarzschild = Spacetime([x0,x1,x2,x3], [M])
        sage: schwarzschild.metric[0,0] = -1+2*M/x1
                # definition of the other components of the metric
```

write_equations(spacetime,coordevs,filename="eqfunc.f90")

```
Calculates the geodesic equations and writes them to a Fortran module

    Parameters
    ----------
    spacetime : Spacetime()
        Spacetime object from which the equations are calculated.
    coordevs : list of sage symbols
        List of the symbols that represent the derivatives of the coordinates
    filename : string, optional
        Name of the file that contains the source code generated. The default name is
"eqfunc.f90".

    Warnings
    --------
    Note that Fortran is case-insensitive, so care must be taken in the variable names used.

    Example
    -------
    After creating a spacetime object (for example, 'schwarzschild') and defining the
components of the metric,
    sage: u0,u1,u2,u3 = var('u0,u1,u2,u3')
    sage: write_equations(schwarzschild,[u0,u1,u2,u3],filename='file.f90')
```

How they work, in a nutshell

A metric tensor is created via FiniteRankFreeModule (requires Sage >= 7.3). The 'coordinates' list is used for partial derivatives, and the 'parameters' list is not used yet, at this stage is only used for defining the variables in the Fortran output code. The 'write_equations()' function first calls internal functions that do the calculations, and then converts everything to Fortran and outputs it.

Let us examine some of the internal functions called by 'write_equations()':

- christoffel_comp(spacetime,i,j,k) : it calculates one component of the Christoffel symbol Γ^i_{jk}, using the equation $\Gamma^i_{jk} = \frac{1}{2} g^{im} \left(\partial_k g_{mj} + \partial_j g_{mk} - \partial_m g_{jk} \right)$.
- generate_equations(spacetime,coordevs) : it calculates the right-hand side of the geodesic equations (returns a list of expressions) $\frac{du^i}{dl} = -\left(\sum_{ij} \right) \Gamma^i_{jk} u^j u^k$ for $i, j, k = 0,1,2,3$. The equations for $u^i = \frac{dx^i}{dl}$ are added later.

Compiling the generated equations

The resulting Fortran code has the following structure:

```
module geodesics
       implicit none
       real(kind=8) :: rs  definition of parameters
       save   necessary to make the parameters available in the rest of the module
       contains
              function metric(x)  numeric evaluation of the metric
                   returns a matrix
              function geod_equations(x,u) numeric eval. of the right-hand side of the geod. eq.
                   returns a list
       include 'ujti/normalize_u.f90' normaliz. for null geodesics
       include 'ujti/geod_sph.f90'


       include 'ujti/lens.f90'   lenses
       include 'ujti/gredsh.f90' grav. redshift
end module
```

This module has to be compiled with `f2py` (part of Numpy/Scipy):

```
f2py -c -m eqschw eq-schw.f90
```

where `eqschw` is the name of the module to be used in Python, and `eq-schw.f90` is the output Fortran code (generated geodesic equations).

Once compiled, the module can be easily handled with Python.

Gravitational lenses

We now go more in depth on how to set up a Python script that calculates the gravitational lenses, and outputs two matrices that make a transformation from a non-lensed image to a lensed one.

Inside the source code of `lens.f90` we find a subroutine that calculates the final position of a photon from its initial position and direction.

```
lens_sph_rkf(x3s,psi0,xi0,om,od,rmin,rmax,dlmin,dlmax,tol,verbose,res)
     Calculates the last angular position of a photon with the RKF method.
     Spherical coordinates

     Parameters
     ----------
     x3s : initial 3D position of the observer in spherical coordinates
     psi0,xi0 : initial angles of the photon (obs. coord.)
     om : angular frequency of the photon (usually 1)
     od : direction of the observer (cart.)
     rmin : minimum value of r allowed. Negative values of rmin calculate
            the event horizon or surface of the object automatically with
            the condition  abs(g(0,0)) = abs(rmin)
     rmax : maximum value of r allowed (how far the ray gets)
     dlmin : minimum value of the affine parameter interval
     dlmax : maximum value of the affine parameter interval
     tol : tolerance (Runge-Kutta Fehlberg)
     verbose: integer that controls the level of output in the program
           0 : prints nothing
           1 : prints the affine parameter l, position and velocity
               in spherical coordinates
           2 : prints the position in cartesian coordinates
     res : output variable (array)
```

Let us explore in more detail some of these options:

- The initial position `x3s` is the position from the origin to the observer ("camera").
- `od` is the direction to which the camera is looking.
- `psi0,xi0` are the angular coordinates of the photon, seen from the observer. We can interpret it as the initial "pixel position" in angular coordinates.
- The second and third entries of the array `res` gives us the last "pixel position" of the photon, as seen through the camera. In fact, this result is then used by another subroutine, `pixel_pos()`, to calculate the actual pixel coordinate to be used in the image file. The first entry of `res` only serves as a flag to check if the execution ended successfully.
- The `verbose` option helps to debug and observe the actual ray. When the number 2 is specified, the output can be sent to a datafile and easily plotted with Gnuplot (`splot 'data.dat'`) to reveal the geodesic.
- The `rmin` option lets us specify the inner boundary of the calculation, which should correspond to the surface of the compact object or the event horizon of the black hole. Since (for weak fields) $g_{00} \approx -1 - 2\Phi$ where Φ is the potential, $g_{00} = \text{const}$ are equipotential surfaces. The hydrostatic condition for any object implies that its surface be an equipotential.

The helper subroutine that calculates to which pixel (for a image file manageable with Numpy) corresponds a set of coordinates `psi,xi` (in our case, the initial and final positions corresponding to the original image and the lensed image).

```
subroutine pixel_pos(psi1,psi,psi2,xi1,xi,xi2,shape_x,shape_y,res)
       psi1,psi2 : range of space (horiz.) that is covered by the image
       xi1, xi2 : range of space (vert.) that is covered by the image
       psi,xi: position in space to be transformed to a pixel position
       shape_x : number of pixels of the image in the horizontal direction
       shape_y : number of pixels of the image in the vertical direction
       res :  array of two entries containing the result (coordinates of the pixel) in the
order x,y.
```

It is important to notice that these subroutines only calculate one pixel of the lensed image. To form an image, these subroutines must be looped in an interval.

Generation of the transformation matrices

As a minimal example, let us discuss how to set up a lens calculation for the Schwarzschild spacetime and save the resultant transformation matrices. First, we import Numpy and the module `eqschw` we generated previously:

```
import numpy as np
import eqschw as equations
```

The parameters of the metric are set (e.g., $R_S = 1$):

```
equations.geodesics.rs = 1.0
```

and also the parameters of the simulation

```
freq = 1.0
rmin = -0.2
lmax = 100.0
rmax = 20
dlmin = 1e-5
dlmax = 0.1
tol = 1e-10
```

The parameter `freq` is proportional to the frequency of the photon. It can be set to anything, but the parameters involving the affine parameter are then affected.

We set the observer to the cartesian coordinates `(5,0,0)`. The subroutine to be called needs the position in spherical coordinates, so we ned to convert first:

```
obs_pos = np.zeros(3)
xi = 5
yi = 0
zi = 0
```

```
obs_pos[0] = np.sqrt(xi*xi + yi*yi + zi*zi)
obs_pos[1] = np.arccos(zi/c[0])
obs_pos[2] = np.arctan2(yi,xi)
```

The camera is looking towards the compact object, so the observer's direction is defined as

```
od = np.array([-1,0,0])
```

As an example, consider an image to be lensed. We want the simulation to extend between the following angles:

```
psi1 = -np.pi/2+0.5
psi2 = np.pi/2-0.5
xi1 = -np.pi/6-0.1
xi2 = np.pi/6+0.1
```

To avoid distortions in the image, we automatically adjust the intervals. We set the width of the image to 300px, and initialize the matrices:

```
num_pix_x = 300
delta_psi = (psi2-psi1)/num_pix_x
num_pix_y = int((xi2-xi1)/delta_psi)
matrix_x = np.zeros((num_pix_y,num_pix_x),dtype=np.int32)
matrix_y = np.zeros((num_pix_y,num_pix_x),dtype=np.int32)
```

Now, we loop for both intervals, and put the results into two matrices:

```
for i in np.arange(psi1,psi2,delta_psi):
    for j in np.arange(xi1,xi2,delta_psi):
       res = equations.geodesics.lens_sph_rkf(c,i,j,freq,od,rmin,rmax,dlmin,dlmax,tol,0)
       pix_in_coord=equations.geodesics.pixel_pos(psi1,i,psi2,xi1,j,xi2,num_pix_x,num_pix_y)
       pix_end_coord = equations.geodesics.pixel_pos(psi1,res[1],psi2,\
                       xi1,res[2],xi2,num_pix_x,num_pix_y)
       matrix_x[ pix_in_coord[1], pix_in_coord[0] ] = pix_end_coord[1]
       matrix_y[ pix_in_coord[1], pix_in_coord[0] ] = pix_end_coord[0]
```

The matrices can be saved with

```
np.save('matrix_x',matrix_x.astype(int))
np.save('matrix_y',matrix_y.astype(int))
```

Plotting the lensed image

To visualize the lensed image, we can make another script, `plot_lens.py`. Here is an example:

First, import Numpy, Scipy image tools and Matplotlib:

```
import numpy as np
from scipy import misc
```

```
import matplotlib.pyplot as plt
```

Load the matrices:

```
matrix_x = np.load('matrix_x.npy')
matrix_y = np.load('matrix_y.npy')
```

Read the non-lensed image and convert it to grayscale. Resize it to the exact values calculated before

```
f1 =  misc.imread('img.png',flatten=True)
f1 = misc.imresize(f1,(174,300))
```

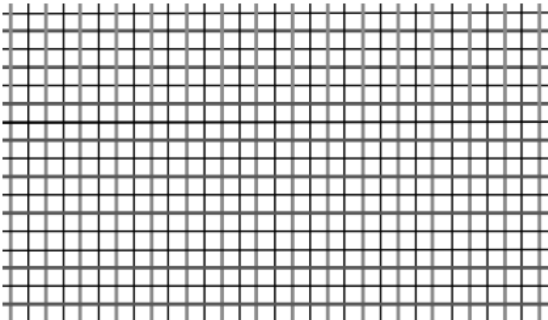
Observe that Scipy demands the size of the image given in the order `(num_pix_y,num_pix_x)`.
Transform the image with the matrices:

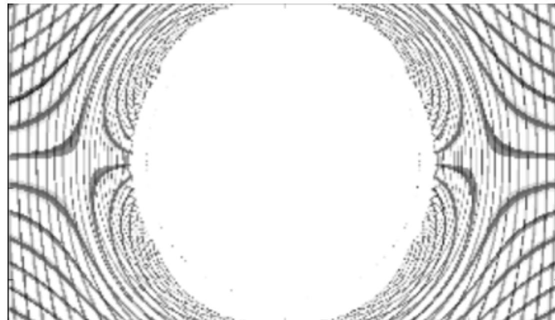
```
f2 = np.zeros(f1.shape)
for i in range(f2.shape[0]):
    for j in range(f2.shape[1]):
        f2[ i,j ] = f1[ matrix_x[i,j], matrix_y[i,j] ]
```

Finally, we can show the image:

```
fig = plt.figure()
img = plt.imshow(f2, cmap=plt.cm.gray)
```

The results for this example are reproduced next.


Original image


Lensed image