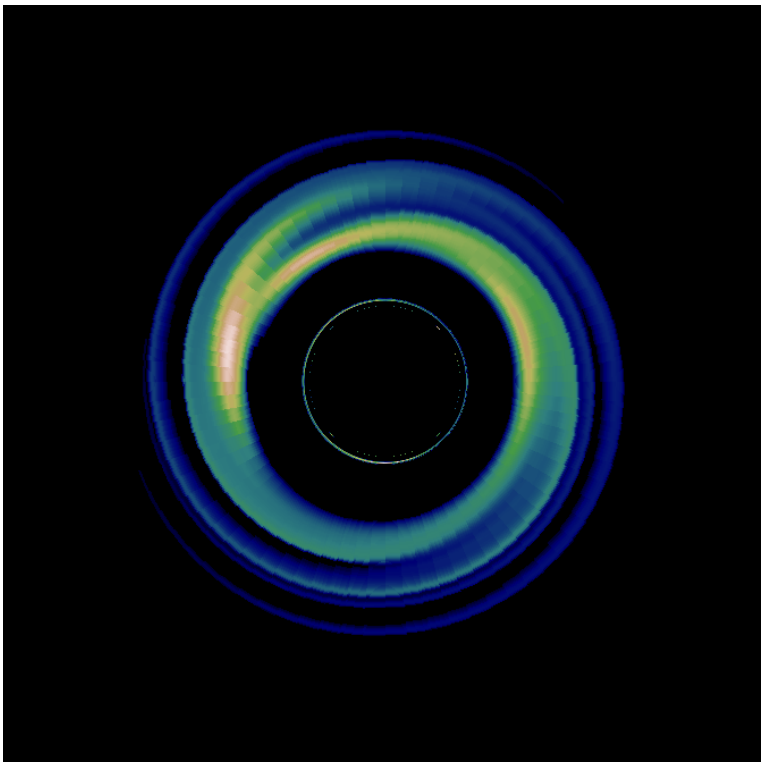
Quick User Guide for GYOTO

Updated October 29, 2019



Introduction - Scope of this Guide

This Guide aims at giving a general presentation of the *General relativitY Orbit Tracer of Observatoire de Paris*, GYOTO (pronounced [dʒɪoto], as for the Italian trecento painter Giotto di Bondone). This text is not a lecture on ray-tracing techniques and is only devoted to presenting the code so that it can be quickly handled by any user. Readers interested in the physics behind GYOTO are referred to Vincent et al. (2011, 2012, and references therein). The aim of this Guide is also to present the code in sufficient details so that people interested to develop their own version of GYOTO can do it easily.

GYOTO is an open source C++ code with a Yorick plug-in that computes null and time-like geodesics in the Kerr metric as well as in any metric computed within the framework of the 3+1 formalism of general relativity. This code allows to compute mainly images and spectra of astrophysical sources emitting electromagnetic radiation in the vicinity of compact objects (e.g. accretion disks or nearby stars).

As GYOTO is continually evolving, this guide will (hopefully) be regularly updated to present the new functionalities added to the code. However, this guide does not constitute a full reference. The reference manual is built from the C++ header files using `doxygen` into the `doc/html/` directory of the source tree. It is also available online (rebuilt every night) at `http://gyoto.obspm.fr/`.

The reader is strongly encouraged to give feedback on this Manual, report typos, ask questions or suggest improvements by sending an email to `frederic.vincent@obspm.fr`

Contents

1 Installing GYOTO

GYOTO is freely available at the URL http://gyoto.obspm.fr/. This URL hosts the online manual of GYOTO, with installation instructions and brief descriptions of the code architecture.

GYOTO is version-controlled with the git software that you should install on your machine. Before uploading the code, be sure that the xerces-c3 (or xercesc3 depending on the architecture) and cfitsio libraries are installed on your system: GYOTO won't compile without these. It is also better (but not required) to install the udunits2 library. Once this is done, just type on a command line

```
git clone git://github.com/gyoto/Gyoto.git
```

which will create a Gyoto repository. It contains directories bin, lib, include, doc, yorick containing respectively the core code and executable, the .C source files, the .h headers, the documentation and Yorick plug-in related code.

In the Gyoto repository, use the standard

```
./configure; make; sudo make install
```

commands to build the code.

In case of problems, have a look at the INSTALL file that gives important complementary informations on how to install GYOTO.

2 Basic usage

2.1 Using the gyoto command-line tool

The most basic way of using GYOTO is through the gyoto command-line tools. It relies on two kinds of files: an XML file containing the inputs and a FITS file containing the outputs.

2.1.1 XML input file

You can find examples of XML input files in doc/examples/. Let us consider the computation of the image of a standard Page-Thorne accretion disk in Boyer-Lindquist coordinates, described in example-page-thorne-disk-BL.xml.

If you are not familiar with XML language, just remember that an XML file is made of several fields beginning with the <Field Name> and ending with </Field Name>. One field can have sub-fields, defined with the same symbols. For instance in example-page-thorne-disk-BL.xml, there is one global field, Scenery, describing the scenery that will be ray-traced, with a few sub-fields: Metric describing the metric used for the computation, here the Kerr metric in Boyer-Lindquist coordinates; Screen describing the observer's screen properties; finally Astrobj describing the astrophysical object that will be ray-traced, here a Page-Thorne accretion disk. All the parameters in this input file can be changed to specify a new scenery.

Let us present in details the example-page-thorne-disk-BL.xml file:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Scenery>
```

The following lines specify the metric: it is the Kerr metric, expressed in Boyer-Lindquist coordinates, with spin 0 (so the Schwarzschild metric here!):

```
  <Metric kind = "KerrBL">
    <Spin>
      0.
    </Spin>
  </Metric>
```

The metric is now defined, let us describe the observer's screen. The `Position` field gives the screen's 4-position in Boyer-Lindquist coordinates (t, r, θ, φ), angles in radians, time and radius in geometrical units (i.e. units with c and G put to 1). The `Time` field gives the time of observation. The `FieldOfView` is given in radians. The screen's `Resolution` is the number of screen pixels in each direction.

```
  <Screen>
    <Position>
      1000.
       100.
       1.22
       0.
    </Position>
    <Time unit="geometrical_time">
       1000.
    </Time>
    <FieldOfView>
       0.31415926535897932384626433832795028841971 6
     </FieldOfView>
     <Resolution>
       32
     </Resolution>
  </Screen>
```

The screen is now defined. The following line describes the target object that will be ray-traced:

```
   <Astrobj kind = "PageThorneDisk"/>
```

Here the target object is very simple and requires no specifications. The `Scenery` is now fully defined and the field can be closed

```
  </Scenery>
```

This is the end of the `XML` input file!

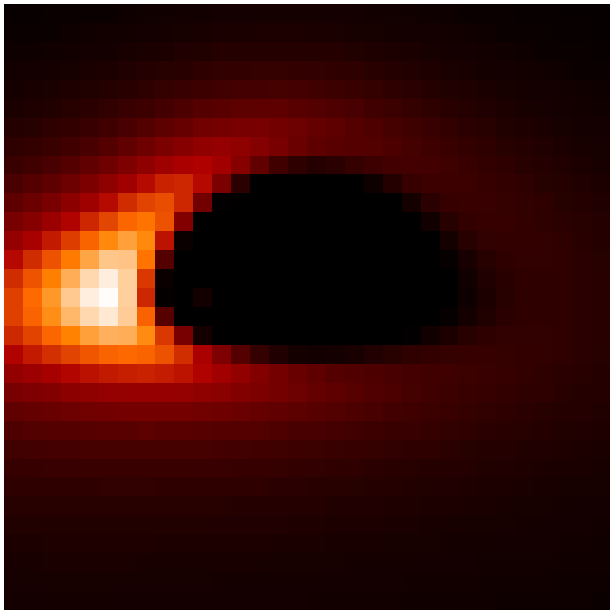


Figure 1: Image of a Page-Thorne thin accretion disk around a Schwarzschild black hole, with a 32×32 pixels screen.

2.1.2 Calling `gyoto`

We will now use the `gyoto` command-line tools to ray-trace the scenery described in this `XML` file. The command-line options are documented in the usual UNIX-style manpage:

```
$ man gyoto
```

Once the `XML` input file is ready, the call to `gyoto` is done according to the following line:

```
$ gyoto input.xml \!output.fits
```

where `input.xml` is the above mentioned `XML` file and `output.fits` is the name of the `FITS` that will contain the result of the computation. The `!` before the `.fits` file allows to overwrite a pre-existing file. If you remove it, an error will occur if the `.fits` file already exists.

The line above asks `GYOTO` to integrate a null geodesic from each pixel of the screen backward in time towards the astrophysical object.

2.1.3 FITS output file

Once the computation is performed, the `output.fits` file is created. You can visualise it by using the `ds9` software (`http://hea-www.harvard.edu/RD/ds9/site/Home.html`) and simply running:

```
$ ds9 ouput.fits
```

For instance, if you use the `example-page-thorne-disk-BL.xml` as is, you will obtain Fig. 1.

2.2 Parallelisation

Ray-tracing of several hundreds of light-rays is a problem that is easily parralelised, by letting different CPUs compute distinct geodesics. GYOTO offers several facilities to perform such parallelisation, depending on the hardware and software environment.

2.2.1 Multi-threading

You can accelerate computations by using several cores on a computer using the `--nthreads=NCORES` option. `NCORES` is the number of threads that GYOTO will use. The optimal value is usually the number of hardware CPU cores on the machine. This option can also be specified in the input file using the <NThreads> entity. This facility does not work for LORENE-based metrics (see Sect. 7).

2.2.2 Multi-processing

GYOTO is able to use the Message Passing Interface (MPI) to distribute the workload over many CPUs, possibly hosted on different computers. YOu can activate it by specifying `-nprocesses=NPROCS`. `NPROCS` is the number of helper processes that GYOTO will spawn. This does not include the main GYOTO process, which will act as a manager for the helpers. This functionnality relies on Astrobj::fillElement() and Metric::fillElement() being properly implemented, which is not always the case for new classes. Also, classes that use supplemental data (additional files referenced to in the XML file) do require that these supplemental data be accessible to all the processes using the same absolute path. Most notably, Lorene metrics require such data. Astrobj classes such as the PatternDisk also require on-disk data.

2.2.3 Poor-mans parallelisation

Another cheap way of parallelising the computation is to call several `gyoto` instances, running on different CPUs or even on different machines, each instance computing only a portion of the image. This sort of basic parrallelisation is, naturally, supported by all the GYOTO metrics.

You can ask GYOTO to compute only a fraction of the screen's pixels by running one of:

```
$ gyoto -iIMIN:IMAX:DI -jJMIN:JMAX:DJ input.xml \!output.fits
$ gyoto --ispec=IMIN:IMAX:DI --jspec=JMIN:JMAX:DJ input.xml \!output.fits
$ gyoto --imin=IMIN --imax=IMAX --jmin=JMIN --jmax=JMAX --di=DI --dj=DJ \
          input.xml \!output.fits
```

where `IMIN`, `IMAX`, `JMIN`, `JMAX` are the extremal indices of the pixels that will be computed. `DI` and `DJ` are the step size in the i and j direction respictively. With the `--ispec` or `-i` syntax, `IMAX` defaults to `IMIN` if there is no colon in the specification, and to the image resolution otherwise. For instance, to compute only the geodesic that hits the central pixel of a 32×32 screen, type one of:

```
$ gyoto -i16 -j16 input.xml \!output.fits
$ gyoto --ispec=16 --jspec=16 input.xml \!output.fits
$ gyoto --imin=16 --imax=16 --jmin=16 --jmax=16 input.xml \!output.fits
```

To compute only the points with even i and od j, use (for instance) one of:

```
$ gyoto -i2::2 -j::2 input.xml \!output.fits
$ gyoto --ispec=2::2 --jspec=1::2 input.xml \!output.fits
$ gyoto --imin=2 --di=2 --jmin=1 --dj=2 input.xml \!output.fits
```

To compute only the lower-right quadrant of the image:

```
$ gyoto -i17: -j:16 input.xml \!output.fits
$ gyoto --ispec=17: --jspec=:16 input.xml \!output.fits
$ gyoto --imin=17 --jmax=16 input.xml \!output.fits
```

How to recombine the several output files into a single FITS file is left as an exercise to the reader. It is easily done using any scientific interpreted language such as Yorick[1] or Python[2].

2.3 The `gyotoy` interface

The second most basic tool provided by `GYOTO` is `gyotoy` (Fig. 2). This is a graphical user interface to visualize a single geodesic. See the `README` and `INSTALL` files for the prerequisites. Once the installation is complete, your launch gyotoy as:

```
$ gyotoy
```

or, from the `yorick/` sub-directory of the built source tree:

```
$ ./yorick -i gyotoy.i
```

followed, on the Yorick prompt, by:

```
> gyotoy
```

You can select a `KerrBL` metric and set the spin, or any other metric defined in an `XML` file. As of writing, `gyotoy` assumes that the coordinate system is spherical-like. It should work in Cartesian coordinates as well, but the labels will be odd. It is possible to select which kind of geodesic to compute (time-like or light-like, using the `Star` or `Photon` radio buttons), the initial position and 3-velocity of the particle, and the projection (a.k.a. the position of the observer). The bottom tool bar allows selecting a few numerical parameters such as whether or not to use an adaptive step. Menus allow saving or opening the parameters as an XML file, exporting the geodesic as a text file, and saving the view as an image file in various formats. The view can be zoomed and panned by clicking in the plot area.

3 Beyond the basics: scripting `GYOTO`

We have seen the two most basic ways of using `GYOTO`: computing a single frame using the `gyoto` command-line tool, and exploring a single geodesic using the `gyotoy` interface. There is much more that `GYOTO` can be used for: computing spectra, performing model-fitting, computing movies, evaluating lensing effects etc.

[1] `http://dhmunro.github.io/yorick-doc/`
[2] `https://www.python.org/`

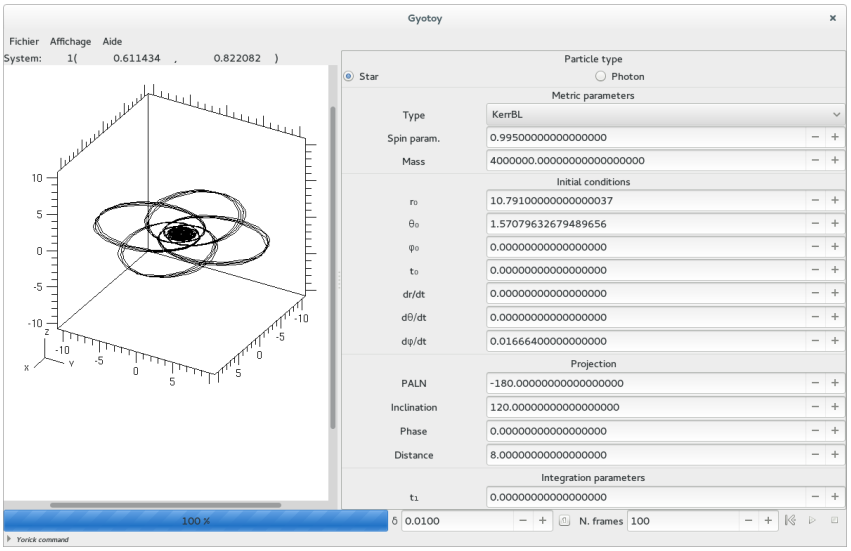


Figure 2: The `gyotoy` graphical user interface.

3.1 Using the Python module

The Python module is constantly evolving and some details can change in future releases. It is documented in a Pythonic way, so help(object) is your friend.

The Python module is split into several submodules:

gyoto.core expose the generic framework compiled in `libgyoto`;

gyoto.std expose the derived classes compiled in the standard plug-in, `libgyoto-stdplug`;

gyoto.lorene expose the derived classes compiled in the Lorene plug-in, `libgyoto-lorene`;

gyoto.metric, gyoto.astrobj, gyoto.spectrometer, gyoto.spectrum regroup the various Metrics, Astrobjs, etc. from the various extensions;

gyoto.util contains a few high-level wrappers and helper functions;

gyoto.animate contains a frame-work for producing videos based on Gyoto.

3.1.1 Building and installing

How to build and install these extensions is documented in INSTALL. At the moment, this is not done automatically. The requisites are:

- Python 3, including the development files;
- Swig (tested with 2.0.12 and 3.0.2);
- NumPy, installed with its development files for the specific Python interpreter you plan on using.

For instance, on Debian Buster, to use compile the `gyoto` extension for Python 3.7, you need the packages:

- `python3.7-dev`;
- `python3-numpy`;
- `swig` or `swig2.0`.

Configure the Gyoto source tree specifying the Python interpreter (if you don't want to use the default on your system), build and install gyoto, then move to the `python/` subdirectory, build and install:

```
$ ./configure PYTHON=/usr/bin/python3.7
$ make
$ sudo make install
$ cd python
$ make
$ sudo make install
```

Depending on your system, you may need to add the directory where the `gyoto/` directory containing `__init__.py` has been installed to your PYTHONPATH variable.

3.1.2 Using

Some sample code can be found in `python/example.py`. The Python extension matches the C++ API very closely, so the C++ reference in `doc/html/` is quite relevant. Most of it can also be accessed through the Python function '`help`'.

The `gyoto.core` module should be enough to perform ray-tracing on anykind of objects, even located in plug-ins:

Importing `gyoto`, `gyoto.core` or `gyoto.std` would normally load the standard plug-in for you, but this is how you would do it manually:

```
import gyoto.core
gyoto.core.requirePlugin("stdplug")
```

Get help on Gyoto:

```
help(gyoto.core)
```

Read scenery from XML:

```
a=gyoto.core.Factory("../doc/examples/example-moving-star.xml")
sc=a.scenery()
```

or:

```
import gyoto.util
sc=gyoto.util.readScenery("../doc/examples/example-moving-star.xml")
```

Create Astrobj by name, access property by name:

```
tr=gyoto.core.Astrobj('Torus')
tr.set('SmallRadius', 0.5)
```

To access methods of specific derived classes (for instance the Star API, which allows computing time-like geodesics), the python extension for a specific Gyoto plug-in must be imported:

```
from gyoto import std
st=std.Star()
```

The various submodules and extensions are fully compatible with each other:

```
sc=gyoto.core.Scenery()
st=std.Star()
sc.astrobj(st)
```

Pointers to the base classes can be up-cast to derived classes:

```
ao = sc.astrobj() # ao contains a Gyoto::Astrobj::Generic * pointer
st = std.Star(ao) # if conversion fails, error is thrown
```

The source directory contains several examples and test cases. The Python source for gyoto.util and gyoto.animate also provide rich examples, always more up-to-date than the present documentation.

3.2 Using the Yorick plug-in

Warning: the Yorick plug-in will not be udated with new features anymore and we plan on phasing it out has soon as gyotoy has been ported to Python. If in doubt, use Python instead.

Yorick is a fairly easy to learn interpreted computer language. We provide a Yorick plug-in which exposes the `GYOTO` functionalities to this language. This plug-in is self documented: at the Yorick prompt, try:

```
> #include "gyoto.i"
> help, gyoto
```

A lot of the `GYOTO` test suite is written in Yorick, which provides many example code in the various `*.i` files in the `yorick/` directory of the `GYOTO` source tree. Another example is provided by the `gyotoy` graphical interface (Sect. 2.3).

For Yorick basics, see:

- `https://github.com/dhmunro/yorick`;
- `http://dhmunro.github.io/yorick-doc/`;
- `http://yorick.sourceforge.net/`;
- `http://www.maumae.net/yorick/doc/index.php`.

As a very minimalist example, here is how to ray-trace an XML scenery into a FITS file in Yorick:

```
$ rlwrap yorick
```

This launches Yorick within the line-editing facility `rlwrap` (provided separately). Then, from the Yorick prompt:

```
> #include "gyoto.i"
> restore, gyoto;
> sc = Scenery("input.xml");
> data = sc(,,"Intensity");
> fits_write, "output.fits", data;
```

or, in two lines:

```
> #include "gyoto.i"
> fits_write, "output.fits", gyoto.Scenery("input.xml")(,,"Intensity");
```

Likewise, to integrate the spectrum over the field-of-view:

```
> #include "gyoto.i"
> restore, gyoto;
> sc = Scenery("input.xml");
> data = sc(,,"Spectrum[mJy.pix-2]");
> spectrum = data(sum, sum, );
> freq = sc(screen=)(spectro=)(midpoints=, unit="Hz");
> plg, spectrum, freq;
> xytitles, "Frequency [Hz]", "Spectrum [mJy]";
```

MPI multi-processing is also available from within Yorick. To activate this functionality, you must call `gyoto.mpiInit` early in your script. Spawn the required number of processes using the `mpispawn` method, and don't forget to use the `mpiclone` method. Although not strictly necessary, it is also recommended to explicitely terminate helper processes that have been spawned in the background (using `mpispawn=0`, and to call `gyoto.mpiFinalize` at the very end of your script:

```
> #include "gyoto.i"
> restore, gyoto;
> // call MPI_Init():
> mpiInit;
> sc = Scenery("input.xml");
> // spawn helpers and send them the scenery
> sc, mpispawn=12, mpiclone=;
> // compute stuff
> data =sc();
> // shut down the helper processes
> sc, mpispawn=0;
> // shut down MPI
> mpiFinalize;
> quit;
```

The Yorick plug-in is not generated automatically. Therefore only a subset of the Gyoto API is exposed. However, all the properties of any object (most of what can be set in XML) can be read and written from Yorick, with the possibility of specifying a unit for properties that support it:

```
> #include "gyoto.i"
> restore, gyoto;
> tr = Astrobj("Torus");
> noop, tr.SmallRadius(0.5, unit="geometrical");
> small_radius_in_meters=tr.SmallRadius(unit="m");
> tr, SmallRadius=0.5, unit="geometrical", LargeRadius=5e6, unit="m";
> large_radius_in_default_unit=tr(LargeRadius=);
```

3.3 Other languages

The Python extension (Sect. 3.1) is generated automatically using the Swig tool, with only little python-specific code. It should therefore be rather easy to compile extensions for the other languages that Swig supports (Tcl, java, R, and many others). If you want it to happen, feel free to contact the Gyoto developers.

3.4 Interfacing directly to the `GYOTO` library

The core functionality is provided as a C++ shared library, which is used both by the `gyoto` command-line tool and the Yorick plug-in. You can, of course, interface directly to this library. The reference is generated from the source code using `doxygen` in the directory `doc/html/`. The application binary interface (ABI) is likely to change with every commit. We try to maintain a certain stability in the application programming interface (API), and to maintain a stable branch whic honly sees bug-fixes between official releases. But the effort we put into this stability is function of the needs of our users. If you start depending on the `GYOTO` library, please contact us (`gyoto@sympa.obspm.fr`): we will try harder to facilitate your work, or at least warn you when there is a significant API change.

4 Choosing the right integrator

Numerical ray-tracing can be very much time-consuming. In order to control the numerical errors in your application, it is wise to experiment with the numerical tuning parameters. `GYOTO` provides several distinct integrators (depending on compile-time options):

- the `Legacy` integrator, the first to have been introduced;
- Boost[3] integrators from the `odeint`[4] library. In `GYOTO`, they are called `runge_kutta_*` (see below).

The integrator and its numerical tuning parameters can be specified in either of these three XML sections:

`Scenery` to specify the integrator and parameters used during ray-tracing by the individual photons;

[3] `http://www.boost.org/`

[4] `http://www.boost.org/doc/libs/1_55_0/libs/numeric/odeint/doc/html/index.html`

`Photon` if the XML file describes a single photon (the Yorick plug-in and the `gyotoy` tool can make use of such an XML file);

`Astrobj` if it is a `Star`, for specifying the integrator and parameters used to compute the orbit of this star.

The full set of tuning parameters that *may be* supported by the `Scenery` section is:

```
<Scenery>
  <Integrator> runge_kutta_fehlberg78 </Integrator>
  <AbsTol> 1e-6 </AbsTol>
  <RelTol> 1e-6 </RelTol>
  <DeltaMax> 1.79769e+308 </DeltaMax>
  <DeltaMin> 2.22507e-308 </DeltaMin>
  <DeltaMaxOverR> 1.79769e+308 </DeltaMaxOverR>
  <MaxIter> 100000 </MaxIter>
  <Adaptive/>
  <Delta unit="geometrical"> 1 </Delta>
  <MinimumTime unit="geometrical_time"> -1.7e308 </MinimumTime>
</Scenery>
```

Integrator the integrator to use (one of the runge_kutta_* or Legacy; default: if compiled-in, `runge_kutta_fehlberg78`);

AbsTol [5] absolute tolerance for adapting the integration step (see `http://www.boost.org/doc/libs/1_55_0/libs/numeric/odeint/doc/html/boost_numeric_odeint/odeint_in_detail/generation_functions.html`);

RelTol [5] relative tolerance for adapting the integration step (idem);

DeltaMax [6] the absolute maximum value for the integration step (defaults to the largest possible non-infinite value);

DeltaMin [6] the absolute minimum value (defaults to the smallest possible strictly positive value);

DeltaMaxOverR [6] this is $h = \delta_{\max}/r$ such that, at any position, the integration step may not be larger than $h \times r$ where r is the current distance to the centre of the coordinate system (defaults to 1).

MaxIter maximum number of integration steps (per photon);

Adaptive (or NonAdaptive) whether or not to use an adaptive step;

Delta integration step, initial in case it is adaptive. Not very important, but should be commensurable with the distance to the `Screen` (i.e. don't use Delta= 10^{-6} if the screen is at 10^{13} geometrical units!). Delta can be specified in any distance-like or time-like unit.

[5] The `Legacy` integrator does not support the AbsTol nor the RelTol parameters

[6] The `Legacy` integrator take the DeltaMin, DeltaMax and DeltaMaxOverR parameters in the `Metric` section.

MinimumTime stop integration when the photon travels back to this date. Defaults to the earliest possible date. Can be specified in any time-like or distance-like unit.

4.1 The Boost integrators

If `GYOTO` was compiled with a C++11-capable compiler and with the Boost library (version 1.53 or above), then the following integrators are available:

- `runge_kutta_cash_karp54`;
- `runge_kutta_fehlberg78`;
- `runge_kutta_dopri5`;
- `runge_kutta_cash_karp54_classic` (alternate implementation of `runge_kutta_cash_karp54`).

Those integrators are implemented in the `Worldline` object. This has the advantage that, when ray-tracing the image of a moving star (`Star` class), the star can use a different integrator than the photons. These integrators support all of the parameters described above.

4.2 The `Legacy` integrator

The `Legacy` integrator is a home-brewed 4^{th}-order, adaptive-step Runge–Kutta integrator. It is always available, independent of any compile-time options. It does not support AbsTol nor RelTol, and takes DeltaMin, DeltaMax and DeltaMaxOver in the `Metric` section, not in the `Scenery` or `Astrobj` section. It is not possible to use different tuning parameters for the Star and the Photons if both use the `Legacy` integrator.

The `Legacy` integrator is implemented in the `Metric` object and may be re-implemented by specific `Metric` kinds. Most notably, the `KerrBL` metric reimplements it (this specific implementation takes advantage of the specific constants of motion). When a metric reimplements the `Legacy` integrator, it is possible to choose which implementation to choose by specifying either <`GenericIntegrator`> or <`SpecificIntegrator`> in the <`Metric`> section. The `KerrBL` specific implementation of the `Legacy` integrator accepts one additional tuning parameter: `DiffTol`, which defaults to 10^{-2} and empirically seems to have very little actual impact on the behaviour of the integrator.

4.3 Integrator comparison

It is advisable to try the various integrators in your context, and to play with the tuning parameters. As a rule of thumbs, if you need to change DeltaMin, DeltaMax, DeltaMaxOverR, or MaxIter, it probably means that you should change for a higher-order integrator. The highest order integrator is currently `runge_kutta_fehlberg78` and is the default.

As an example, we have compared all the integrators for a simple situation: the `Metric` is a Schwarzschild black-hole of $4 \times 10^6\,M_\odot$, the `Screen` is at 8 kpc, a single light-ray is launched $50\,\mu$as from the centre, we integrate the light-ray (back in time) during twice

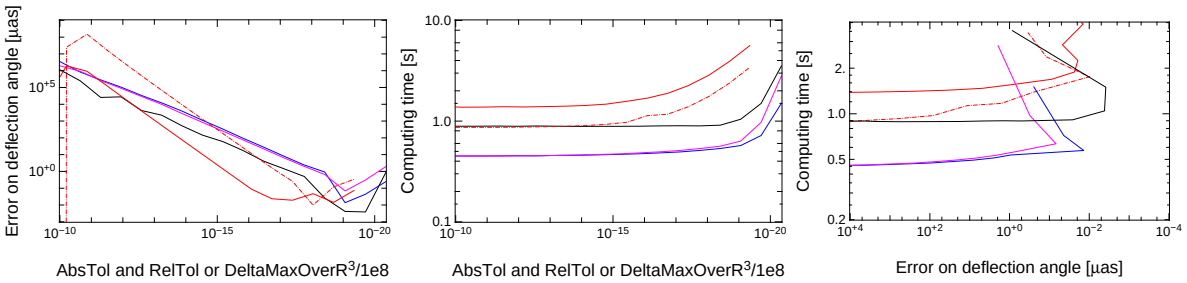


Figure 3: Comparison of the integrators when measuring a deflection angle (see text). *Left:* error on deflection angle vs. tuning parameter; *centre:* computing time vs. tuning parameter; *right:* computing time vs. error on deflection angle. Colours denote the integrator: *red:* `Legacy` (*solid:* specific `KerrBL` implementation, *dash-dotted:* generic implementation); *black:* `runge_kutta_fehlberg78`; *blue:* `runge_kutta_karp54`; *magenta:* `runge_kutta_dopri5`.

the travel-time to the origin of the coordinate centre and compute the deflection angle. We do that for each integrator and a set of numerical tuning parameters. For the `Legacy` integrator, we try both the generic integrator and the specific `KerrBL` implementation, and use DeltaMaxOverR as tuning parameter. For the Boost integrators, we use AbsTol and RelTol as tuning parameters (they are kept equal to each-other). We then compare the result for two values of the tuning parameter and measure the computing time required for each realisation.

For each integrator, there exists an optimum value or the tuning parameter, for which the (estimated) uncertainty in deflection angle is minimal (Fig. 3, left panel). As long as the tuning parameter is larger than (or equal to) this optimum, computation time varies very little (central panel). However, when the tuning parameter becomes too small, numerical error and computation time explode (right panel).

The two low-order Boost integrators are the fastest, the two `Legacy` integrators are the slowest. The default, `runge_kutta_fehlberg78`, has intermediate performance, but seems to yield the smallest error and to be less sensitive on the exact value of the tuning parameter close to its optimum.

All the integrators except the generic implementation of the `Legacy` integrator agree to within $2\,\mu$as on the deflection angle, which is of order 33Â°. The relative uncertainty is therefore of order 10^{-11}.

In conclusion, for this specific use case, the best choice seems to be:

```
<Scenery>
 <Integrator>runge_kutta_fehlberg78</Integrator>
 <AbsTol>1e-19</AbsTol>
 <RelTol>1e-19</RelTol>
</Scenery>
```

If computation time is more critical than accuracy, the other Boost integrators are also good choices.

The Yorick code that was used to generate Fig. 3 is provided in the `GYOTO` source code as `yorick/compare-integrators.i`. You can run it from the top directory of the built source tree as:

```
yorick/yorick -i yorick/compare-integrators.i
```

5 GYOTO architecture

5.1 GYOTO base classes

GYOTO is basically organised around 8 base classes (see Fig. 4):

- The Metric class: it describes the metric of the space-time (Kerr in Boyer-Lindquist coordinates, Kerr in Kerr-Schild coordinates, metric of a relativistic rotating star in 3+1 formalism...).
- The Astrobj class: it describes the astrophysical target object that must ray-traced (e.g. a thin accretion disk, a thick 3D disk, a star...).
- The Spectrum class: it describes the emitted spectrum of the Astrobj.
- The Worldline class: it gives the evolving coordinates of a time-like or null geodesic. The Star class is a sub-class of Worldline as for the time being a star in GYOTO is only described by the time-like geodesic of its centre, with a given fixed radius.
- The WorldlineIntegState class: it describes the integration of the Worldline in the given Metric.
- The Screen class: it describes the observer's screen, its resolution, its position in space-time, its field of view.
- The Scenery class: it describes the ray-tracing scene. It is made of a Metric, a Screen, an Astrobj and the quantities that must be computed (an image, a spectrum...).
- The Factory class: it allows to translate the XML input file into C++ objects.

Fig. 4 presents the main GYOTO classes and their hierarchy.

5.2 A typical GYOTO computation

Let us now describe the main interactions of these various classes during the computation of one given photon, ray-traced in the Kerr metric in Boyer-Lindquist coordinates towards, for instance, a PageThorneDisk, i.e. a geometrically thin optically thick disk following Page and Thorne (1974).

All directories used in the following are located in GYOTO home directory.

GYOTO main program is located in bin/gyoto.C. This program first interprets the command line given by the user. It creates a new Factory object, initialised by means of the XML input file, that will itself create (see lib/Factory.C) the Scenery, Screen and Astrobj objects. The output quantities the user is interested in (image, spectrum...) will be stored during the computation in the data object, of type Astrobj::Properties. The Scenery object is then used to perform the ray-tracing, by calling its rayTrace function.

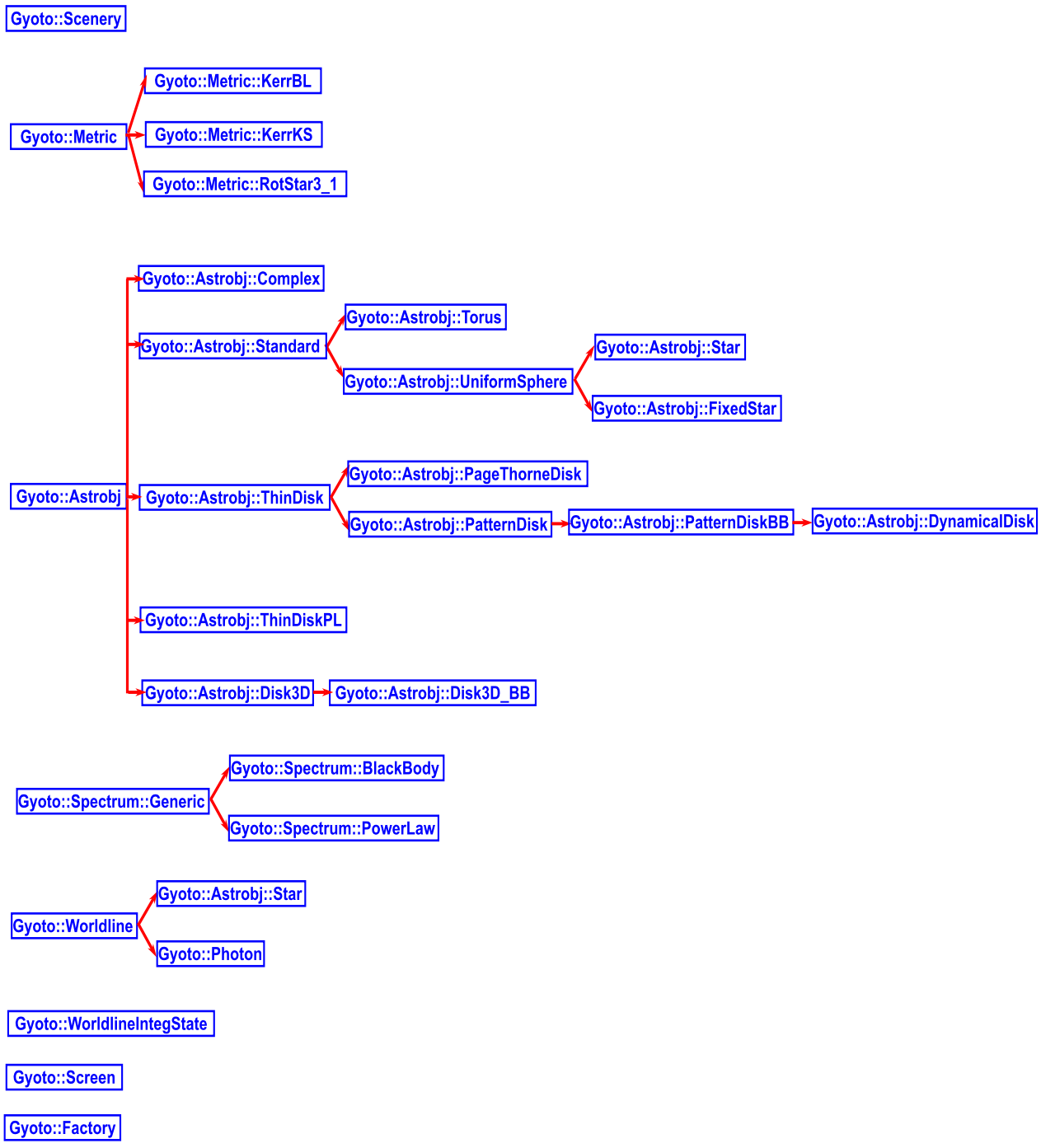


Figure 4: Hierarchy of GYOTO C++ main classes.

All the functions that begin with `fits_` allow to store the final output quantities in `FITS` format.

The function `Scenery::rayTrace` calls `Photon::hit` that forms the core of `GYOTO`: `Photon::hit` integrates the null geodesic from the observer's screen backward in time until the target object is reached (there are other stop conditions of course, see `lib/Photon.C`). `Photon::hit` is basically made of a loop that calls the function `WorldlineIntegState::nextStep` until a stop condition is reached. `WorldlineIntegState::nextStep` itself calls the correct adaptive fourth order Runge-Kutta integrator (RK4) , depending on the metric used. Here, the metric being `KerrBL`, the RK4 used is `KerrBL::myrk4_adaptive`.

Moreover, the `Photon::hit` also calls the `Astrobj::Impact` function that asks the `Astrobj` whether the integrated photon has reached it or not. When the photon has reached the target, this `Astrobj::Impact` function calls the `Astrobj::processHitQuantities` function that updates the `data` depending on the user's specifications.

6 Computing an image or a spectrum in the Kerr metric with GYOTO

6.1 The Screen

The observer's `Screen` is made of $N \times N$ pixels, and has a field of view of f radians. The field of view is defined as the maximum angle between the normal to the screen and the most tangential incoming geodesic.

Keep in mind that the screen is point-like, the different pixels are all at the same position (the one and only screen position defined in the `XML` file), but the various pixels define various angles on the observer's sky. For instance, if $f = \pi/2$ (which gives a view of the complete half space in front of the screen), the geodesic that hits the screen on the central pixel ($i = N/2, j = N/2$) comes from the direction normal to the screen whereas the geodesic that hits the screen on the ($i = N, j = N/2$) pixel comes from a direction tangential to the screen.

Each pixel of the screen thus covers a small solid angle on the observer's sky. This elementary solid angle is equal to the solid angle subtended by a cone of opening angle f divided by the number of pixels: $\delta\Omega_{\rm pixel} = 2\,\pi\,(1 - \cos f)/N^2$ (assuming the field of view is small enough).

6.2 Computing an image

The quantity that is carried along the geodesics computed by `GYOTO` in most cases is the specific intensity I_ν ($\mathrm{erg\,cm^{-2}\,s^{-1}\,ster^{-1}\,Hz^{-1}}$).

An image is then defined as a map of specific intensity: each pixel of the screen contains one value of I_ν, that can then be plotted. It is important to keep in mind that such an "image" is not physically equivalent to a real image that would be obtained with a telescope: a real image is a map of specific fluxes values, and a specific flux is the sum of the specific intensity on some solid angle.

An example of image computation has already been given in Section 2.1.

6.3 Computing a spectrum

To compute a spectrum, the `Screen` field of the `XML` file should contain information about the observed frequency range. For a real telescope, this means adding a spectrometer. The additional command in the `XML` file is thus:

```
<Spectrometer kind="freqlog" nsamples="20">5. 25.</Spectrometer>
```

This line means that that 20 values of observed frequencies will be considered, evenly spaced logarithmically, between 10^5 and 10^{25} Hz. It is possible to choose frequencies linearly evenly spaced by using `freq` instead of `freqlog`. It is also possible to use wavelengths instead of frequencies, see `GyotoScreen.h` for more information.

Moreover, the `XML` file should explicitly state that the quantity of interest is no longer a simple image, but a spectrum. This is allowed by the following command, that should be added for instance before the end of `Scenery` field:

```
<Quantities>Spectrum</Quantities>
```

When this command is used, the output `FITS` file will contain a cube of images, each image corresponding to one observed frequency.

Computing the spectrum is now straightforward. Remembering that the flux is linked to the intensity by:

$$\mathrm{d}F_{\nu_{\mathrm{obs}}} = I_{\nu_{\mathrm{obs}}} \cos\theta \, \mathrm{d}\Omega, \tag{1}$$

where Ω is the solid angle under which the emitting element is seen, and θ is the angle between the normal to the observer's screen and the direction of incidence, the flux is given by:

$$F_{\nu_{\mathrm{obs}}} = \sum_{\mathrm{pixels}} I_{\nu_{\mathrm{obs}},\mathrm{pixel}} \cos(\theta_{\mathrm{pixel}}) \, \delta\Omega_{\mathrm{pixel}}, \tag{2}$$

where $I_{\nu_{\mathrm{obs}},\mathrm{pixel}}$ is the specific intensity reaching the given pixel, θ_{pixel} is the angle between the normal to the screen and the direction of incidence corresponding to this pixel and $\delta\Omega_{\mathrm{pixel}}$ is the element of solid angle introduced above.

This quantity $F_{\nu_{\mathrm{obs}}}$ can thus be very simply computed from the cube of specific intensities computed by `GYOTO`. Examples of spectra computed by `GYOTO` can be found in, e.g., Straub et al. (2012). Section 3.2 contains example code to compute a spectrum directly using the provided Yorick interface. See also `yorick/check-polish-doughnut.i`, which does just that as part of the routine test suite of `GYOTO`.

7 `GYOTO` in numerical metrics

A specificity of `GYOTO` is its ability to ray-trace in non-Kerr metrics, numerically computed in the framework of the 3+1 formalism of general relativity (Gourgoulhon 2012), e.g. by means of the open source LORENE library developed by the Numerical Relativity group at Observatoire de Paris/LUTH[7].

[7] `http://www.lorene.obspm.fr/`

For the time being, only a simple example of numerical metric is implemented in the public version of GYOTO: the metric of a relativistic rotating star. The file doc/examples/example-movingstar-rotstar3_1.xml allows to ray-trace a basic GYOTO moving Star in this metric. The file resu.d specified in the XML file is the output of a LORENE computation for the metric of a rotating relativistic star by the LORENE/nrotstar code.

The basic functions developed in lib/RotStar3_1.C are similar to their Kerr counterparts, but here the metric is expressed in terms of the 3+1 quantities (lapse, shift, 3-metric, extrinsic curvature). The equation of geodesics expressed in the 3+1 formalism is given in Vincent et al. (2012) and implemented in lib/RotStar3_1.C. However, it is possible to choose in the XML file whether the integration will be performed by using this 3+1 equation of geodesics, or by using the most general 4D equation of geodesics (see Vincent et al. 2011, for a comparison of the two methods).

8 Local tetrads: setting the Screen orientation

The orientation of the camera (implemented by the class Gyoto::Screen in C++ and gyoto.core.Screen in Python) is determined by a local tetrad of the observer composed of four 4-vectors $(\vec{u}, \vec{e}_l, \vec{e}_u, \vec{e}_f)$ which form an orthonormal local basis and where:

$\vec{u}$ is the 4-velocity of the observer;

$\vec{e}_l$ points to the left-hand side of the observer;

$\vec{e}_u$ points up;

$\vec{e}_f$ points front, i.e. in the direction where the observer is looking.

In the following, $(\vec{e}_t, \vec{e}_r, \vec{e}_\theta, \vec{e}_\varphi)$ is the usual spherical local basis, which can also be defined for Cartesian coordinate systems (t, x, y, z) such as KerrKS as follows:

$$\begin{aligned} r &= \sqrt{x^2+y^2+z^2} \\ r_p &= \sqrt{x^2+y^2} \\ \theta &= \mathrm{atan2}(r_p, z) \\ \varphi &= \mathrm{atan2}(y, x) \\ \vec{e}_t &= (1,0,0,0) \\ \vec{e}_r &= (0, x/r, y/e, z/r) \\ \vec{e}_\theta &= (0, \cos\theta\cos\varphi, \cos\theta\sin\varphi, -\sin\theta) \\ \vec{e}_\varphi &= (0, -\sin\varphi, \cos\varphi, 0) \end{aligned}$$

The Screen interface provides several ways to specify this tetrad with the concept of ObserverKind (XML entity), which can be set using the Screen method observerKind():

ObserverAtInfinity the default, does no special treatment, which implicitly assumes $(\vec{u}, \vec{e}_l, \vec{e}_u, \vec{e}_f) = (\vec{e}_t, -\vec{e}_\varphi, -\vec{e}_\theta, -\vec{e}_r)$. In this case, the Screen orientation can be fine-tuned using the XML entities PALN, Dangle1 and Dangle2, corresponding to the methods PALN(), dangle1() and dangle2(). This default is fine for most cases, i.e. when the observer really is far from the the central object (more than a few 10 geometrical units) and at rest.

`ZAMO` is the natural extension of `ObserverAtInfinity` in stronger field. $\vec{u}$ is first computed using the Metric method `zamoVelocity` which returns $\vec{e}_t$ projected in the hyperplane orthogonal to $\vec{e}_\varphi$ and then normalized. This is the velocity of a zero-angular-momentum observer. Then, the base $(\vec{u}, -\vec{e}_\theta, -\vec{e}_r, -\vec{e}_\varphi)$ is orthonormalized into $(\vec{u}, \vec{e}_u, \vec{e}_f, \vec{e}_l)$ using the Gram–Schmidt process as implemented in the Metric method `GramSchmidt`. The order of this orthonormalization matters to get the same tetrad as propsed by Krolik, Hawley & Hirose (2005, ApJ:622:1008), implemented in `KerrBL`.

`VelocitySpecified` applies the same process but using a user-supplied, arbitrary 4-velocity using the XML entity `FourVel` or the Screen method `fourVel`.

`FullySpecified` is the most versatile, since the user provides the four vectors using the XML entities `FourVel` ($\vec{u}$), `ScreenVector1` ($\vec{e}_l$), `ScreenVector2` ($\vec{e}_u$) and `ScreenVector3` ($\vec{e}_f$) (the corresponding methods are the same, starting with first letter in lower case). The user is then responsible for providing a direct, orthonormal basis. For constructing this basis, the user can use the helper functions described below.

In the rest of this section, we will use the Python API to document and demonstrate the use of the various helper methods that can be used to construct a local tetrad. Those lines can be translated to C++ in a trivial manner. Let `MetricKind` be a Metric kind (e.g. `KerrBL`, `KerrKS` or other), `AstrobjKind` be a kind of Astrobj, and $x0, x1, x2, x3$ be coordinates in this Metric.

```
from gyoto import core
import numpy
from matplotlib import pyplot as plt

metric=core.Metric('KerrBL')        # make and initialize a metric

scr=core.Screen()                   # make a Screen
scr.metric(metric)
scr.resolution(32)
scr.fieldOfView(60, 'degree')
scr.anglekind('Rectilinear')

ao=core.Astrobj('ThinDisk')         # make an astrobj,
ao.metric(metric)                   # initialize it
ao.set('InnerRadius', 6.)
ao.rMax(200)

sc=core.Scenery()                   # make Scenery
sc.metric(metric)                   # assign metric
sc.astrobj(ao)                      # assign astrobj
sc.screen(scr)                      # assign screen

x0, x1, x2, x3=0., 15., numpy.pi/4, numpy.pi/4
pos=numpy.asarray([x0, x1, x2, x3]) # this is a 4 position
scr.setObserverPos(pos)             # set observer position
fourvel=numpy.zeros(4)              # NumPy array for 4-velocity
```

Now we can set the observer kind and ray-trace. The default would be:

```
scr.observerKind('ObserverAtInfinity')
results=sc.rayTrace()
```

which will not work that close to the central object (15 geometrical units in this example).

```
scr.observerKind('ZAMO')
results=sc.rayTrace()
```

would always work, though. ZAMO stands for 'zero angular momentum observer'. One can display the intensity image:

```
plt.imshow(results['Intensity'], origin='bottom')
plt.show()
```

To try `ObserverKind==VelocitySpecified`, we need to get a 4-velocity. Several methods can help with that (see for instance `help(core.Metric.circularVelocity)`), but to continue with the same example we shall retrieve the velocity of a ZAMO with:

```
metric.zamoVelocity(pos, fourvel)
```

Once the velocity vector has been set, one just needs to assign it to the Screen:

```
scr.observerKind('VelocitySpecified')
scr.fourVel(fourvel)
results=sc.rayTrace()
plt.imshow(results['Intensity'], origin='bottom')
plt.show()
```

`zamoVelocity` is always defined. It provides a velocity with zero angular momentum. `KerrBL.zamoVelocity` is an optimised implementation, compatible with the generic one which projects $\vec{e}_t$ on the hyperplane orthogonal to $\vec{e}_\varphi$. The same could be implemented in Python like this:

```
ephi    = numpy.asarray([0., 0., 0., 1.])
fourvel = numpy.asarray([1., 0., 0., 0.])

if metric.coordKind()==core.GYOTO_COORDKIND_SPHERICAL:
    ephi    = numpy.asarray([0., 0., 0., 1.])
else:
    phi=numpy.arctan2(pos[2], pos[1]);
    cp=numpy.cos(phi)
    sp=numpy.sin(phi)
    ephi=numpy.asarray([0., -sp, cp, 0.])

metric.projectFourVect(pos, fourvel, ephi)
metric.multiplyFourVect(fourvel, 1./abs(metric.norm(pos, fourvel)))
```

The snippet above has introduced three methods:

```
metric.multiplyFourVect(fourvel, scalar)
```

just multiplies each element of fourvel by `scalar`. This C++ convenience method does the same as the more Pythonic:

```
fourvel *= scalar
```

where `fourvel` is a NumPy array and `scalar` a... scalar. Likewise, for adding two vectors, this:

```
metric.addFourVect(fourvel, v)
```

is equivalent to this:

```
fourvel += v
```

where `fourvel` and `v` are two NumPy arrays of 4 elements.

```
metric.norm(pos, fourvel)
```

`norm` returns the norm of a 4-vector: $\|\vec{v}\| = \text{sgn}(\vec{v} \cdot \vec{v})\sqrt{|\vec{v} \cdot \vec{v}|}$. It could also be computed as:

```
n2=metric.ScalarProd(pos, fourvel, fourvel)
numpy.sign(n2)*numpy.sqrt(abs(n2))
```

where `ScalarProd` implements the scalar product.

```
metric.projectFourVect(pos, fourvel, ephi)
```

projects `fourvel` on the hyperplane orthogonal to `ephi` and returns the projected vector in place. The mathematical equivalent would be

$$\text{proj}_{\vec{v}}(\vec{u}) = \vec{u} - \text{sgn}(\|v\|)(\vec{u} \cdot \frac{\vec{v}}{|||\vec{v}|||})\frac{\vec{v}}{|||\vec{v}|||}$$

`VelocitySpecified` instructs the Screen to build a local tetrad for a camera looking towards the origin. The same can be done with `FullySpecified` like this:

```
scr.observerKind('FullySpecified')
el=numpy.zeros(4)
eu=numpy.zeros(4)
ef=numpy.zeros(4)
metric.observerTetrad(pos, fourvel, el, eu, ef)
scr.fourVel(fourvel)
scr.screenVector1(el)
scr.screenVector2(eu)
scr.screenVector3(ef)
results=sc.rayTrace()
plt.imshow(results['Intensity'], origin='bottom')
plt.show()
```

If `fourvel` has been set using `metric.zamoVelocity(pos, fourvel)`, this actually is equivalent to setting `ObserverKind` to `'ZAMO'`.

The method `gyoto.animate.orbiting_screen_forward` provides a complete example where the local tetrad is computed in Python, orthonormalized using `metric.GramSchmidt` and passed to the Screen where the property `ObserverKind` has been set to `FullySpecified`. We comment on a simplified version below.

The goal of this code is to set the camera such that it looks in the direction where the Screen is moving, so $\vec{e}_f$ is initialized as the 3-velocity $(0, \dot{x}^1/\dot{x}^0, \dot{x}^2/\dot{x}^0, \dot{x}^3/\dot{x}^0)$. $\vec{e}_u$ is initialized as $\mathrm{proj}_{e_f}(\vec{e}_r)$ and normalized. It is therefore a vector pointing away from the central object, orthogonal to e_f and normal (in the 3D Euclidian sense). We set $\vec{e}_l = \vec{e}_u \times \vec{e}_f$ where $\times$ denotes the 3D external (vector) product. Thus, $(\vec{e}_l, \vec{e}_u, \vec{e}_f)$ would form an orthonormal basis of a Euclidian 3D space. We then use the Gram–Schmidt process to orthonormalize the tetrad $(\vec{u}, \vec{e}_u, \vec{e}_f, \vec{e}_l)$ in the metric sense. Most computations are done in Cartesian 3D coordinates, for the sake of computing the external product easily.

```
pos=numpy.asarray([x0, x1, x2, x3]) # this is a 4 position
fourvel=numpy.zeros(4)
metric.circularVelocity(pos, fourvel)

# pos and fourvel have been initialized previously
coord=numpy.concatenate((pos, fourvel))
ef=numpy.zeros(4)
# get Cartesian expression of 3-velocity into ef[1:]:
metric.cartesianVelocity(coord, ef[1:])
# normalize ef to 1 (in 3D):
ef /= numpy.sqrt((ef*ef).sum())

# get Cartesian expression of position in posc
# as well as spherical radius in r:
if metric.coordKind()==core.GYOTO_COORDKIND_SPHERICAL:
    t=pos[0]
    r=pos[1]
    theta=pos[2]
    phi=pos[3]
    st=numpy.sin(theta)
    ct=numpy.cos(theta)
    sp=numpy.sin(phi)
    cp=numpy.cos(phi)
    posc=[pos[0], r*st*cp, r*st*sp, r*ct]
else:
    posc=pos
    r=numpy.sqrt(pos[1:]**2).sum()

# Now initialize eu as er, but project it orthogonally to ef in 3D
# and then normalize it:
eu=numpy.concatenate(([0.], posc[1:]/r))
eu -= (ef*eu).sum()*ef
eu /= numpy.sqrt((eu*eu).sum())
```

```
# Last vector is el, which we construct as the 3D vector product
# between eu and ef, computed in Cartesian coordinates:
el=numpy.zeros(4)
el[1]=eu[2]*ef[3]-eu[3]*ef[2]
el[2]=eu[3]*ef[1]-eu[1]*ef[3]
el[3]=eu[1]*ef[2]-eu[2]*ef[1]

# If the coordinate system is spherical, we need to convert back:
if metric.coordKind()==core.GYOTO_COORDKIND_SPHERICAL:
    er=posc/r
    ephi=numpy.asarray([0., -sp, cp, 0.])
    etheta=numpy.asarray([0., ct*cp, ct*sp, -st])
    ef=numpy.asarray([
      0.,
      (er*ef).sum(),
      (etheta*ef).sum(),
      (ephi*ef).sum()
    ])
    eu=numpy.asarray([
      0.,
      (er*eu).sum(),
      (etheta*eu).sum(),
      (ephi*eu).sum()
    ])
    el=numpy.asarray([
      0.,
      (er*el).sum(),
      (etheta*el).sum(),
      (ephi*el).sum()
    ])

# Our 3 vectors form an orthonormal basis in the Euclidian sense,
# but the tetrad still needs to be orthonormalized in the metric
# sense:
metric.GramSchmidt(pos, fourvel, eu, ef, el);

# Then we just need to attach the tetrad to the Screen:
scr.observerKind('FullySpecified')
scr.setObserverPos(pos)
scr.fourVel(fourvel)
scr.screenVector1(el)
scr.screenVector2(eu)
scr.screenVector3(ef)

# Perform ray-tracing as usual:
results=sc.rayTrace()
plt.imshow(results['Intensity'], origin='bottom')
plt.show()
```

9 Extending GYOTO

This section is currently under construction.

GYOTO can be extended easily by adding new Metric, Astrobj, and Spectrum classes. Such classes can be written in the Python language (Sect. 9.1) or in C++, in which case they are made available to Gyoto by cumpiling them into a GYOTO plug-in (Sect. 9.2). The main GYOTO code-base already contain tree plug-ins:

- stdplug, which contain all the standard analytical metrics and all the standard astrophysical object;
- python, which allows implementing new Metric, Astrobj, and Spectrum classes in the Python 3 language;
- and lorene, which contains the numerical, LORENE-based metrics.

In addition, we maintain our own private plug-in, which contains experimental or yet unpublished Astrobj and Metric classes. When we make a publication based on these classes, we try to move them from our private plug-in to the relevant public plug-in. We kindly request that you follow the same philosophy: whenever you write a new class and make a publication out of it, please publish the code as free software, either on your own servers or by letting us include it in GYOTO.

As soon as you write your own objects (especially if you do so in C++), you will dependent on the stability of the GYOTO application programming interface, which is subject to frequent changes. It will help us to help you maintain your code if you inform us of such development: gyoto@sympa.obspm.fr. In addition, we try to maintain a stable branch in our github repository (http://github.com/gyoto/Gyoto). Code on this branch should remain API-and ABI-compatible with the latest official release.

9.1 Extending in Python

The 'python' plug-in allows writting derived classes in Python rather than C++.

This is the easiest approach and lets you set-up your new objects in just a few lines of code, without the need for setting-up a plug-in, compiling it frequently during your deverlopment cycle. However, there is a cost in terms of code execution speed. This code is usually negligible to moderate for classes implementing the Astrobj and Spectrum interfaces, but it is major for classes implementing the Metric interface. This is because the Metric methods are evaluated several times per iteration when iterating geodesics, while the Astrobj and Spectrum methods are evaluated less frequently.

To use your Python classes, simply load the Python plug-in, for instance by setting the GYOTO_PLUGINS environment variable to include the "python"[8] plug-in (see Sect. 9.6). Then tell the plug-in about your module and class. For instance, in an XML file:

[8]The actual name of the Gyoto plug-in will match that of the Python interpreter used for compiling it. For instance, if Gyoto was configured with ./configure PYTHON=/usr/bin/python3.4, the plug-in will be called python3.4 instead of python. This allows you to compile several versions of the plug-in, that will run inside each of the Python interpreters.

```
<Metric kind="Python">
 Python module name for ``my_module.py'':
  <Module>my_module</Module>

 Class defined in ``my_module.py'':
  <Class>my_class</Class>

 Any parameters (foating point) ``my_class'' may need:
  <Parameters>2.0 20.0</Parameters>
</Metric>
```

The Python class will need to implement a few methods, with the same name as the C++ method it implements (except the C++ name `operator()` is translated as `__call__` in Python).

More information can be found in the doxygen-generated documentation for the `GyotoPython.h` file and in the `plugins/python/doc/examples` sub-directory of the source code distribution.

9.2 Writing a C++ plug-in

A plug-in is merely a shared library which contains the object code for your new objects, plus a special initialisation function. It is loaded into memory using `dlopen()` by the function `Gyoto::loadPlugin(char const*const name, int nofail)` (note that the upper level function `Gyoto::requirePlugin(char const*const name, int nofail)` should be used instead whenever applicable), implemented in `lib/Register.C`. The `name` argument will be used three times:

- the shared library file must be called `libgyoto-`*name*`.$suffix` (`$suffix` is usually `.so` under Linux, `.dylib` under Mac OS);
- the init function for your plug-in must be called either `__Gyoto`*name*`Init` or exactly `__GyotoPluginInit`;
- each subcontractor registered by the init function (see below) will be tagged with the plug-in name, so it is later possible to search for a registered subcontractor by kind name and plugin name.

The role of the init function is to register subcontractors for your object classes so that they can be found by name. For instance, assuming you want to bundle the Minkowski metric (actually provided in the standard plug-in) in a plugin called `myplugin`, you would write a C++ file (whose name is not relevant, but assume `MyPlugin.C`) with this content:

```
#include "GyotoMinkowski.h"
using namespace Gyoto;
extern "C" void __GyotomypluginInit() {
  Metric::Register("Minkowski", \&Metric::Subcontractor<Metric::Minkowski>);
}
```

Likewise, you could register more metrics, astrophysical objects and spectra. Other examples can be seen in the `lib/StdPlug.C` and `lib/LorenePlug.C` files, and in the `plugins/python/` directory.

When building your plug-in, make sure `MyPlugin.C` ends up compiled with `lib/Minkowski.C` (in this example) into a single shared library called `libgyoto-myplugin.so` (assuming you work under Linux), and drop this file somewhere where Gyoto will find it at run-time. Gyoto will try to ldopen the plug-ins from the following locations in order (some variables are defined in `gyoto.pc`, see below):

- wherever the run-time linker looks by default, which typically includes:
  - any directory listed in the `$LD_LIBRARY_PATH` environment variable;
  - `/usr/local/lib/`;
  - `/usr/lib/`;
- `${localpkglibdir}/${GYOTO_SOVERS}/`;
- `${localpkglibdir}/` (typically `/usr/local/lib/gyoto`);
- `${GYOTO_PLUGDIR}/`, this is the directory where the standard plug-ins shipped with Gyoto are installed (typically `${pkglibdir}/${GYOTO_SOVERS}/`);
- `${pkglibdir}/` (typically `/usr/local/lib/gyoto` or `/usr/lib/gyoto`).

Note that `${localpkglibdir}` is defined only when it makes sense (i.e. when Gyoto is not itself installed under `/usr/local`).

`GYOTO` ships a pkg-config file (`gyoto.pc`) which stores useful build-time information such as the install prefix and the plug-in suffix. This file gets installed in the standard location, by default `/usr/local/lib/pkgconfig/gyoto.pc`.

The Gyoto source code contains several examples of plug-ins. One of them is minimalistic and its only purpose is to illustrate how to build a plug-in, including an autoconf-based build system that parses `gyoto.pc`. It is a good starting point for writing your own plug-in. See the content of the subdirectory `plugins/null/`, in particular the `README` file.

9.3 Adding a new metric (`Metric`)

The simplest example for a `Metric` object is certainly the `Minkowski` class. Let's go through the header file that defines its interface (expunged from all this useless documentation `;-)`), we trust the reader to go see the corresponding `.C` file:
Avoid multiple and recursive inclusion of header files:

```
#ifndef __GyotoMinkowski_H_
#define __GyotoMinkowski_H_
```

Minkowski is a metric, include the `Metric` base class header file:

```
#include <GyotoMetric.h>
```

Declare that our new class goes into the `Gyoto::Metric` namespace:

```
namespace Gyoto {
  namespace Metric { class Minkowski; }
}
```

Declare that Minkowski inherits from the base class:

```
class Gyoto::Metric::Minkowski
: public Gyoto::Metric::Generic
{
```

Each class must be friend with the corresponding `SmartPointer` class:

```
  friend class Gyoto::SmartPointer<Gyoto::Metric::Minkowski>;
```

Minkowski has no private data, else we would put it here:

```
  private:
```

Every class needs a constructor, which will at least populate the `kind_` attribute of the parent class and the kind of coordinate system (Cartesian-like or spherical-like):

```
public:
  Minkowski();
```

The cloner is important, and easy to implement. It must provide a deep copy of an object instance. It is used in the multi-threading case to make sure two threads don't tip on each-other's toes, and in the Yorick plug-in (Sect. 3.2) when you want to make a copy of an object rather than reference the same object:

```
  virtual Minkowski* clone() const ;
```

Then come the two most important methods, which actually define the mathematical metric:

```
void gmunu(double g[4][4], const double * x) const ;
int christoffel(double dst[4][4][4], const double * x) const ;
```

The `setParameter` method is the one that interprets options from the XML file. For each XML entity found in the `Metric` section in the form
`<ParName unit="unit_name">ParValueString</ParName>`,
the method `Metric::Generic::setParameters()` will call
`setParameter(ParName, ParValueString, unit_name)`:

```
  virtual void setParameter(std::string, std::string, std::string);
```

`setParameter()` should interpret the parameters specific to this class and pass whatever remains to the `Generic` implementation. `setParameter()` has a counterpart, `fillElement()`, which is mostly used by the Yorick plug-in (Sect. 3.2) to dump an in-memory object instance to text format (this is what allows `gyotoy`, Sect. 2.3, to write its parameters to disk). This method must be compiled only if XML input/output is compiled in:

```
#ifdef GYOTO_USE_XERCES
  virtual void fillElement(FactoryMessenger *fmp);
#endif
```

The Minkowski implementation goes on with an alternate implementation of `gmunu()` and `christoffel()`. For the purpose of this documentation, we will skip these additional examples and close the header file here:

```
};
#endif
```

For more details, see the `GYOTO` reference manual in `doc/html/` or at `http://gyoto.obspm.fr/`. There are a few other methods that are worthwhile reimplementing, such as `circularVelocity()`, which allows getting accurate beaming effects for thin disks and tori. Naturally, `circularVelocity()` can only be implemented if circular orbits physically exist in this metric (else, the Keplerian approximation is readily provided by the generic implementation). Some other low-level methods can be reimplemented, but it is not necessarily a good idea.

Once you have implemented the new Metric, just make sure it is compiled into your plug-in and initialised in the initialisation function (Sect. 9.2). For official `GYOTO` code (that does not depend on LORENE), this is done by adding your `.C` file to `libgyoto_stdplug_la_SOURCES` in `lib/Makefile.am` (and running `autoreconf` followed by `configure`), and adding one line in `__GyotostdplugInit` in `lib/StdPlug.C`.

9.4 Adding a new spectrum (`Spectrum`)

Adding a new spectrum kind is almost the same as adding a metric.

9.5 Adding a new astrophysical target object (`Astrobj`)

Adding a new astronomical object kind is almost the same as adding a metric. However, astronomical objects are more complex than metrics (they can have abundant hair). Instead of inheriting directly from the most generic base class, `Gyoto::Astrobj::Generic`, you will save yourself a lot of effort if you are able to derive from one of the higher level bases:

Astrobj::ThinDisk a geometrically thin disk (e.g. `PageThorneDisk`, `PatternDisk`);

Astrobj::UniformSphere a... uniform sphere (e.g. `Star`, `FixedStar`);

Astrobj::Standard any object whose boundary can be defined as an iso-surface of some function of the coordinates, such as a sphere or a torus (e.g. `UniformSphere`, `Torus`).

9.6 Using your new plug-in

There are several ways to instruct `GYOTO` to load your plug-in. You can set the environment variable`GYOTO_PLUGINS`[9], with a command such as

[9]How to do it depends on your shell and is outside the scope of this manual.

```
export GYOTO_PLUGINS=stdplug,myplugin
```

Alternatively, a list of plug-ins can be specified on the command-line when using the `gyoto` tool:

```
$ gyoto --plugins=stdplug,myplugin ...
$ gyoto -pstdplug,myplugin ...
```

A plug-in can also be specified in the input XML file for each section:

```
<Metric kind = "KerrBL" plugin="stdplug">
```

Finally, the Yorick interface (Sect. 3.2) has a function to explicitly load a `GYOTO` plug-in at run-time:

```
> gyoto.requirePlugin("myplugin");
```

Once the plug-in is loaded, your new object kinds should be registered (that's the purpose of the init function). To check that your objects are correctly register, you can use the `--list` (or `-l`) option of the `gyoto` tool or the `gyoto.listRegister()` function of the Yorick interface:

```
$ gyoto --list [input.xml]
$ gyoto -l [input.xml]
```

You can use then use your classes directly in an XML file, using the name you provided to the `Register()` function in the init function of the plug-in:

```
<Metric kind = "Minkowski" plugin="myplugin">
```

The Yorick interface can load any object from an XML description, an can also initialise any object from its name:

```
> metric = gyoto.Metric("Minkowski", "myplugin");
```

If your object supports any options using the `setParameter()` method, these options can also be set from within Yorick:

```
> metric, setparameter="ParName", "ParValueString", unit="unit_string";
```

If your want finer access to your objects from the Yorick interface, you will need to provide a Yorick plug-in around your `GYOTO` plug-in. Look at the content of the `yorick/stdplug/` directory. For new official `GYOTO` objects in the standard plug-in, it is fairly easy to provide an interface directly inside our Yorick plug-in.

9.7 Quality assurance

It is customary to provide a test-suite for every new class in `GYOTO`. This normally includes an example file in `doc/examples`, which is ray-traced in the `check` target of `bin/Makefile.am`. Usually, we also provide a new Yorick test file called `yorick/check-myclass.i` which is called from `yorick/check.i`. This is a good idea to do so even if you don't intend on using the Yorick plug-in: at least, you can use this interpreted interface to perform unit tests on your code in a more fine-grained manner than a full-featured ray-traced image.

References

Gourgoulhon, E.: 2012, *3+1 Formalism in General Relativity*, Springer, Heidelberg, Germany.

Page, D. N. and Thorne, K. S.: 1974, Disk-Accretion onto a Black Hole. Time-Averaged Structure of Accretion Disk, *ApJ* **191**, 499–506.

Straub, O., Vincent, F. H., Abramowicz, M. A., Gourgoulhon, E. and Paumard, T.: 2012, Modelling the black hole silhouette in Sagittarius A* with ion tori, *A&A* **543**, A83.

Vincent, F. H., Gourgoulhon, E. and Novak, J.: 2012, 3+1 geodesic equation and images in numerical spacetimes, *Classical and Quantum Gravity* **29**(24), 245005.

Vincent, F. H., Paumard, T., Gourgoulhon, E. and Perrin, G.: 2011, GYOTO: a new general relativistic ray-tracing code, *Classical and Quantum Gravity* **28**(22), 225011.