A Gyoto Primer

Thibaut Paumard
for the Gyoto team

LESIA

1 June 2016

Gyoto: a framework for...

Means:

- Accessing metric coefficients and Christoffel symbols;
- Integrating geodesics (at least time-like and null);
- Computing quantities along geodesics (in particular radiative transfer).

(Original) Goals:

- Computing stellar orbits;
- Ray-tracing (spectro-)images;
- Visualising space-time.

Gyoto limitations

Originally meant for astrophysics

- 4D (other dimensions would require deep changes);
- Checks for physical relevance (should be easy to deactivate):
  - $v \leq c$ (i.e. no space-like geodesics);
  - no integration inside black-hole event horizon...
- Most high-level features (ray-tracing!) assume:
  - first coordinate is t (metric signature...);
  - three other are Cartesian-like or spherical-like.

(Could be changed, if meaningful...)

Non-astrophysical use

- Integrating geodesics and functions along geodesics, in 4D.

What is ray-tracing? Can it work for you?

Build a picture

- Pin-hole camera: a point in space-time;
- Consider a grid of (null) geodesics that cross this point;
- Integrate some quantity backwards in “time” along each of these geodesics.

A Gyoto Scenery

- Metric: trajectory of Photons;
- Astrobj: astronomical object, matter that interacts with light (emission/absorbtion);
- Screen: pin-hole camera.

Outline

Components

C++ library (libgyoto)

- Generic framework (Factory, Scenery...);
- Base classes (for metrics, astronomical objects...).

C++ plug-ins (libgyoto-stdplug et al.)

Implementation of specific metrics etc.

`gyoto` command-line utility

Read description from XML, ray-trace, save to FITS.

Interpreted, interactive interfaces

- For two languages (ATM): Yorick and Python;
- Complex algorithms made easy (e.g. model fitting);
- Graphical user interface: gyotoy

Important classes in libgyoto

Base classes

Implemented by derived classes in plug-ins:

Metric::Generic The metric at this end of the Universe;

Astrobj::Generic Baryonic matter content of the Universe;

- **Astrobj::Standard** Object defined as $f(x^0, x^1, x^2, x^3) < k$ (e.g. a ball, a torus...);
- **Astrobj::ThinDisk** Equatorial thin disk.

Spectrum::Generic Used by some Astrobj::Generic subclasses;

Spectrometer::Generic Wavelengths to which the camera is sensitive.

Important classes in libgyoto

Generic framework

Screen Location, field-of-view etc. of the camera;

Scenery Top-level class for ray-tracing, contains a Metric::Generic, a Screen, an Astrobj::Generic;

Factory Build object from XML desciption and describe object in XML format;

Worldline Geodesic, either null (Photon) or time-like (Star).

`stdplug`: the standard plug-in

Subclasses of Gyoto::Astrobj::Generic/Standard/ThinDisk

- Astrobj::Complex: container for several Astrobj instances;
- Geometrical models: Astrobj::Star, Astrobj::Torus...;
- Physical models: Astrobj::PolishDoughnut...;
- Models from simulations: Astrobj::Disk3D...

Subclasses of Gyoto::Metric::Generic

- KerrBL (Kerr in Boyer–Lindquist coordinates);
- KerrKS (Kerr in Kerr–Schild coordinates);
- Minkowski (in Cartesian and spherical coordinates).

Subclasses of Gyoto::Spectrum::Generic

- Powerlaw, BlackBody, ThermalBrehmstrahlung

Other plug-ins

`lorene`: interface to... LORENE

Metrics based on LORENE, a library for numerical relativity developped at LUTh.

`obspm`: Paris Observaroty private plug-in

- Where new stuff is developped prior to publication;
- Classes normaly migrate to `stdplug` after publication.

`python`: objects implemented in Python

Objects impremented in Python

- Metric::Generic (`gmunu`, `christoffel`);
- Astrobj::Standard (`__call__`, `getVelocity`) and Astrobj::ThinDisk;
- Spectrum::Generic (`__call__`)

Define a Metric in Python (could use sage!)

my_metric.py (or <InlineModule/> in XML)

```
import numpy
import gyoto
class Minkowski:
    def gmunu(self, g, x):
        for mu in range(0, 4):
            for nu in range(0, 4):
                g[mu][nu]=g[nu][mu]=0
        g[0][0]=-1;
        for mu in range(1, 4):
            g[mu][mu]=1.
    def christoffel(self, dst, x):
        for alpha in range(0, 4):
            for mu in range(0, 4):
                for nu in range(0, 4):
                    dst[alpha][mu][nu]=0.
        return 0
```

The integrators

Several integrators are available

- Specific to a Metric, tuning parameters in the Metric: `Legacy`.
  - 4-th order Runge–Kutta in Metric::Generic;
  - RK4 on optimized equation for KerrBL;
  - Optimized but broken in KerrKS;
  - 3+1 integrator for numerical metrics (see Frédéric's talk).
- Generic for all metrics, boost-based, tuning parameters in the Worldline: `runge_kutta_(cash_karp54|fehlberg78|dopri5)`. Interests:
  - well tested;
  - estimate numerical errors by comparing integrators;
  - usually more accurate, sometimes faster;
  - use distinct tuning parameters for Photons and Stars.

Life-cycle of a Photon (context: ray-tracing)

Integrate backwards in time

- Created by Scenery, using Screen position;
- Pixel position in Screen corresponds to velocity at detection;
- Step (δ) controlled for accuracy, and to not miss Astrobj;
- When Astrobj is hit, let it fill the AstrobjProperties and update transmission (may require refining geodesic);
- Stop conditions:
  - high optical depth;
  - Photon is escaping (actually coming from infinity);
  - photon comes from sink region;
  - too many iterations.

Outline

The gyoto command-line tool

Synopsis

```
$ gyoto [--options] input.xml output.fits
```

Some usual options (see `man gyoto`)

`--plugins=`*`plugin1,plugin2...`* list of plug-ins to load;

`--debug` enable debug output (extremely verbose);

`--imin=`*`ix`* `--imax=`*`i1`* `--di=`*`di`* (same for `j`)
compute only part of the field;

`--time=`*`t`* `--fov=`*`f`* `--resolution=`*`r`* `...`
Screen parameters: observing date, field-of-view...

`--nthreads=`*`nt`* `|` `--nprocesses=`*`np`*
set number of threads or processes;

`--impactcoords[=`*`file.fits`*`]`
save or load impact coordinates.

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

  <Metric kind = "KerrBL">
    <Spin> 0. </Spin>
    <Mass unit="sunmass"> 4e6 </Mass>
  </Metric>

  <Screen>
    <Distance unit="kpc"> 8 </Distance>
    <Time unit="yr"> 30e3 </Time>
    <FieldOfView unit="uas"> 150 </FieldOfView>
    <Inclination unit="degree"> 90 </Inclination>
    <PALN> 0 </PALN>
    <Argument> 0 </Argument>
    <Resolution> 32 </Resolution>
    <Spectrometer kind="wave" nsamples="1" unit="um">
      2.0 2.4
    </Spectrometer>
  </Screen>
```

```xml
  <Astrobj kind = "FixedStar">
    <Radius> 12 </Radius>
    <Position> 0 0 0 </Position>
    <Spectrum kind="PowerLaw">
      <Exponent> 0 </Exponent>
      <Constant> 0.001 </Constant>
    </Spectrum>
    <Opacity kind="PowerLaw">
      <Exponent> 0 </Exponent>
      <Constant> 0.01 </Constant>
    </Opacity>
    <OpticallyThin/>
  </Astrobj>

  <Delta> 1e0 </Delta>
  <MinimumTime> 0. </MinimumTime>
  <Quantities> Spectrum[Jy.sr-1] </Quantities>
  <NProcesses>12</NProcesses>

</Scenery>
```

Yorick: small, elegant, easy

- An interpreted language;
- Simple, elegant C-like syntax;
- Friendly, reliable author;
- Popular among French interferometrists and adaptive optics experts;
- Easy to extend with plug-ins.

Currently used in Gyoto for:

- Graphical user interface: `gyotoy`;
- Regression testing suite;
- Model-fitting algorithms (astrometry, spectra);
- Complex algorithms such as “matte-painting”.

Strength and weaknesses

Hand-written

- ☺ Optimized;
- ☺ Free to adapt syntax;
- ☹ Only middle to high level is exposed;
- ☹ Hard(er and harder) to maintain.

Nice integration of Properties

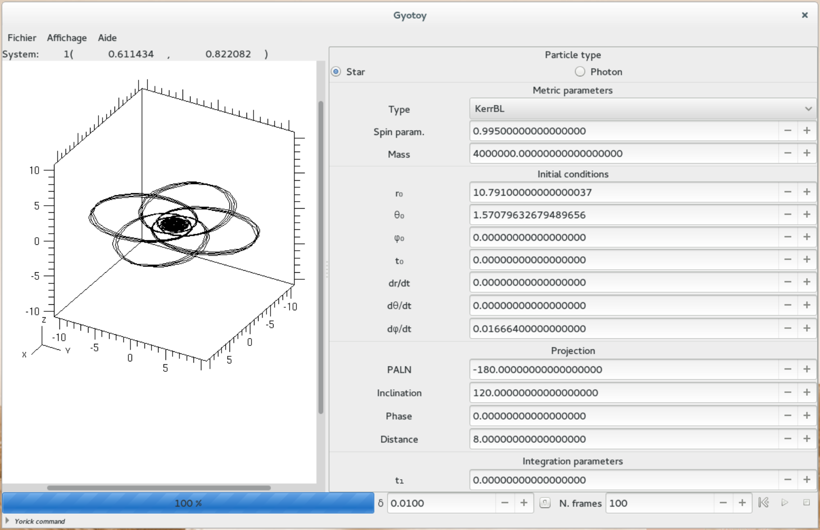
XML:

```
<Metric kind="KerrBL">
  <Mass unit="sunmass"> 4e6 </Mass>
 </Metric>
```

Yorick:

```
metric =
  gyoto.Metric("KerrBL", Mass=4e6, unit="sunmass");
```

gyotoy


Gyotoy
Fichier Affichage Aide
System: 1(0.611434 , 0.822082)
Particle type
Star
Photon
Metric parameters
Type KerrBL
Spin param. 0.99500000000000000
Mass 4000000.0000000000000000
Initial conditions
r0 10.791000000000000037
θ0 1.57079632679489656
φ0 0.00000000000000000
t0 0.00000000000000000
dr/dt 0.00000000000000000
dθ/dt 0.00000000000000000
dφ/dt 0.01666400000000000
Projection
PALN -180.00000000000000000
Inclination 120.00000000000000000
Phase 0.00000000000000000
Distance 8.00000000000000000
Integration parameters
t1 0.00000000000000000
100 %
δ 0.0100
N. frames 100
Yorick command

Python: ubiquitous, powerful

- Very wide user community;
- Many extensions readily available;
- Writting extensions made easy by external tools (Swig).

Fairly new, not yet used for:

- Fine-grained unit tests;
- Model-fitting;
- Complex algorithms requiring other extensions;
- Interfacing with SageMath...

Strength and weaknesses

Mostly automatic (Swig-based)

- ☺ Extensible to other languages (e.g. R, Octave, Scilab);
- ☺ Access to low-level functions, including in plug-ins;
- ☺ Scalable;
- ☹ Bit hard to add higher-level interfaces;
- ☹ More complex, more overheads.

Yorick:

```
metric =
  gyoto.Metric("KerrBL", Mass=4e6, unit="sunmass");
```

Python:

```
metric = gyoto.Metric('KerrBL')
metric.set('Mass', 4e6, 'sunmass')
```

Outline

Computing an orbit

Using the command-line

```
$ gyotoy
```

In Yorick

```
#include "gyoto.i"
restore, gyoto;
metric = KerrBL(Mass=4e6, unit="sunmass",
                Spin=0.995);
star = Star(Metric=metric,
            initcoord=[0., 10.791, pi/2., 0.],
                      [0., 0., 0.016664]);
star, xfill=3000.;
txyz = star.get_txyz();
plg, txyz(,3), txyz(,2);
```

Computing an orbit

In Python

```
# Load extensions
import numpy
import matplotlib.pyplot as plt
import gyoto
loadPlugin("stdplug")
import gyoto_std

# Make metric
metric = gyoto_std.KerrBL()
metric.set('Mass', 4e6, 'sunmass')
metric.set('Spin', 0.995)

# Make star
star = gyoto_std.Star()
star.metric(metric)

# Set initial coordinate
```

Computing an orbit

In Python

```
# Set initial coordinate
pos=array_double(4)
pos[0]=pos[3]=0
pos[1]=10.791
pos[2]=numpy.pi*0.5
vel=array_double(3)
vel[0]=vel[1]=0.
vel[2]=0.016664
star.setInitCoord(pos, vel)

# Integrate
star.xFill(3000.)

# Retrieve computed coordinates
n=star.get_nelements()
t2=numpy.ndarray(n)
r2=numpy.ndarray(n)
```

Computing an orbit

In Python

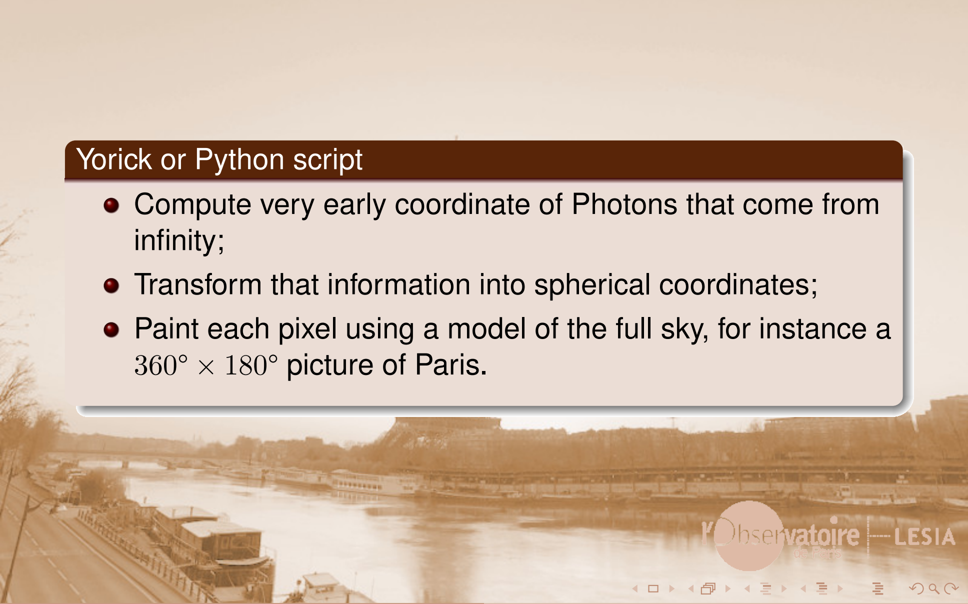
```
# Retrieve computed coordinates
n=star.get_nelements()
t2=numpy.ndarray(n)
r2=numpy.ndarray(n)
theta2=numpy.ndarray(n)
phi2=numpy.ndarray(n)

t=gyoto.array_double.fromnumpy1(t2)
r=gyoto.array_double.fromnumpy1(r2)
theta=gyoto.array_double.fromnumpy1(theta2)
phi=gyoto.array_double.fromnumpy1(phi2)

star.getCoord(t, r, theta, phi)

# Plot, using matplotlib
plt.plot(r2*numpy.cos(phi2), r2*numpy.sin(phi2))
plt.show()
```

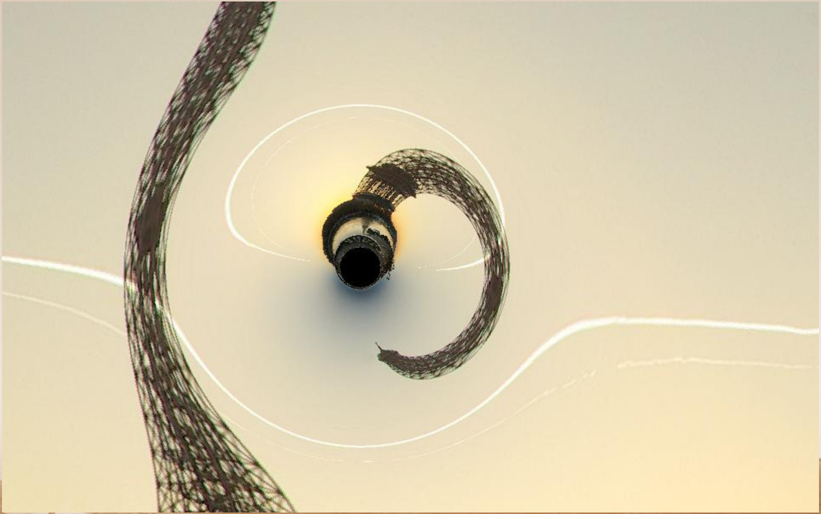
Matte painting



Yorick or Python script

- Compute very early coordinate of Photons that come from infinity;
- Transform that information into spherical coordinates;
- Paint each pixel using a model of the full sky, for instance a $360° \times 180°$ picture of Paris.

Copyrights: 2007 Alexandre Duret-Lutz for the original panorama, 2014 Thibaut Paumard for the derived work presented here. License CC BY-SA 2.0.



Define a Metric in Python

my_metric.py (or <InlineModule/> in XML)

```
import numpy
import gyoto
class Minkowski:
    def gmunu(self, g, x):
        for mu in range(0, 4):
            for nu in range(0, 4):
                g[mu][nu]=g[nu][mu]=0
        g[0][0]=-1;
        for mu in range(1, 4):
            g[mu][mu]=1.
    def christoffel(self, dst, x):
        for alpha in range(0, 4):
            for mu in range(0, 4):
                for nu in range(0, 4):
                    dst[alpha][mu][nu]=0.
        return 0
```

Instanciate it

In Python

```python
import gyoto
gyoto.loadPlugin('python')

gg=gyoto.Metric("Python")
gg.set("Module", "my_metric")
gg.set("Class", "Minkowski")
```

In XML

```xml
<Metric kind = "Python" plugin="python">
  <Mass unit="sunmass"> 4e6 </Mass>
  <Module>my_metric</Module>
  <Class>Minkowski</Class>
</Metric>
```

Conclusion

Gyoto is a great tool

- Works the same in analytical and numerical metrics;
- Is very accurate and fast;
- Is extremely versatile.

Integration with Sage

Gyoto and Sage can already call each other through Python.

- Make sure geodesics can be computed in any 4D metric (independent of signature);
- Make sure any geodesic can be computed independent of physical relevance...;
- Develop one sage-based metric;
- On the longer run, allow other dimensions;
- Anything else?