EinsteinPy

Release 0.4.dev0

unknown

Jan 16, 2021

CONTENTS

EinsteinPy is an open source pure Python package dedicated to problems arising in General Relativity and gravitational physics, such as geodesics plotting for Schwarzschild, Kerr and Kerr Newman space-time model, calculation of Schwarzschild radius, calculation of Event Horizon and Ergosphere for Kerr space-time. Symbolic Manipulations of various tensors like Metric, Riemann, Ricci and Christoffel Symbols is also possible using the library. EinsteinPy also features Hypersurface Embedding of Schwarzschild space-time, which will soon lead to modelling of Gravitational Lensing! It is released under the MIT license.

View source code of EinsteinPy!

Key features of EinsteinPy are:

- Geometry analysis and trajectory calculation in vacuum solutions of Einstein's field equations
- Schwarzschild space-time
- Kerr space-time
- Kerr-Newman space-time
- Various utilities related to above geometry models
- Schwarzschild Radius
- Event horizon and ergosphere for Kerr black hole
- Maxwell Tensor and electromagnetic potential in Kerr-Newman space-time
- And much more!
- Symbolic Calculation of various quantities
- Christoffel Symbols
- Riemann Curvature Tensor
- Ricci Tensor
- Index uppering and lowering!
- Simplification of symbolic expressions
- Geodesic Plotting
- Static Plotting using Matplotlib
- Interactive 2D plotting
- Environment aware plotting!

- Coordinate conversion with unit handling
- Spherical/Cartesian Coordinates
- Boyer-Lindquist/Cartesian Coordinates
- Hypersurface Embedding of Schwarzschild Space-Time
- Shadow cast by an thin emission disk around a Schwarzschild Black Hole

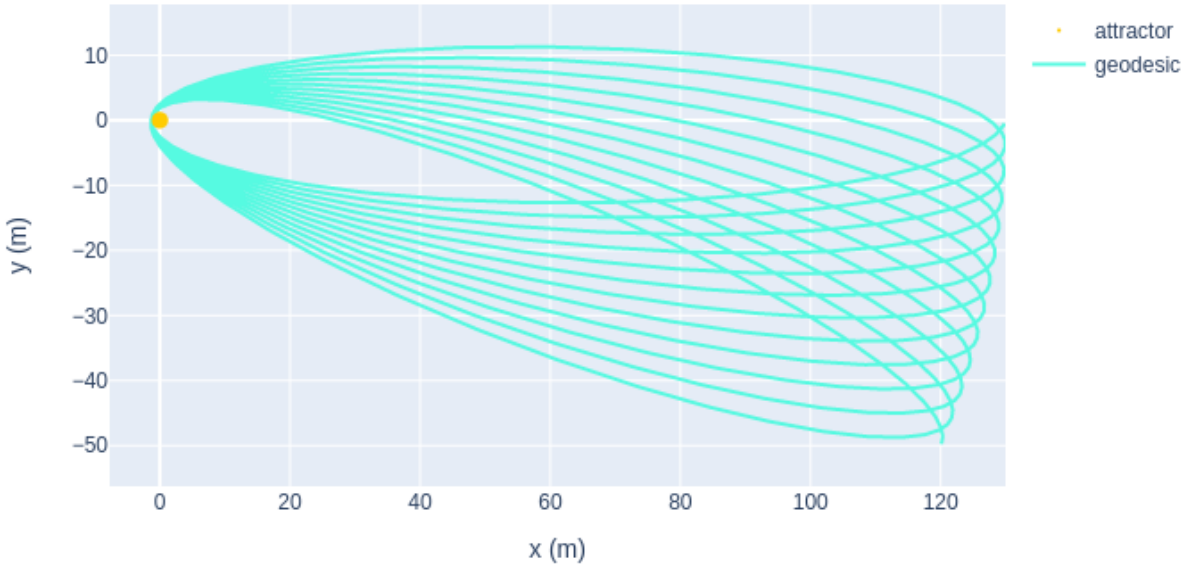
And more to come!

Einsteinpy is developed by an open community. Release announcements and general discussion take place on our mailing list and chat.

The source code, issue tracker and wiki are hosted on GitHub, and all contributions and feedback are more than welcome. You can test EinsteinPy in your browser using binder, a cloud Jupyter notebook server:

EinsteinPy works on recent versions of Python and is released under the MIT license, hence allowing commercial use of the library.

```
from einsteinpy.plotting import GeodesicPlotter
from einsteinpy.examples import perihelion
a = GeodesicPlotter()
a.plot(perihelion())
a.show()
```



CHAPTER

ONE

CONTENTS

1.1 Getting started

1.1.1 Overview

EinsteinPy is a easy-to-use python library which provides a user-friendly interface for supporting numerical relativity and relativistic astrophysics research. The library is an attempt to provide programming and numerical environment for a lot of numerical relativity problems like geodesics plotter, gravitational lensing and ray tracing, solving and simulating relativistic hydrodynamical equations, plotting of black hole event horizons, solving Einstein's field equations and simulating various dynamical systems like binary merger etc.

1.1.2 Who can use?

Most of the numerical relativity platforms currently available in the gravitational physics research community demands a heavy programming experience in languages like C, C++ or their wrappers on some other non popular platforms. Many of the people working in the field of gravitational physics have theoretical background and does not have any or have little programming experience and they find using these libraries mind-boggling. EinsteinPy is motivated by this problem and provide a high level of abstraction that shed away from user all the programming and algorithmic view of the implemented numerical methods and enables anyone to simulate complicated system like binary merger with just 20-25 lines of python code.

Even people who does not know any python programming can also follow up with the help of tutorials and documentation given for the library. We aim to provide all steps, from setting up your library environment to running your first geodesic plotter with example jupyter notebooks.

So now you are motivated enough so let's first start with installing the library.

1.1.3 Installation

It's as easy as running one command!

Stable Versions:

For installation of the latest `stable` version of EinsteinPy:

- Using pip:

```
$ pip install einsteinpy
```

- Using conda:

```
$ conda install -c conda-forge einsteinpy
```

Latest Versions

For installing the development version, you can do two things:

- Installation from clone:

```
$ git clone https://github.com/einsteinpy/einsteinpy.git
$ cd einsteinpy/
$ pip install .
```

- Install using pip:

```
$ pip install git+https://github.com/einsteinpy/einsteinpy.git
```

Development Version

```
$ git clone your_account/einsteinpy.git
$ pip install --editable /path/to/einsteinpy[dev]
```

Please open an issue here if you feel any difficulty in installation!

1.1.4 Running your first code using the library

Various examples can be found in the examples folder.

1.1.5 Contribute

EinsteinPy is an open source library which is under heavy development. To contribute kindly do visit :

https://github.com/einsteinpy/einsteinpy/

and also check out current posted issues and help us expand this awesome library.

1.2 User guide

1.2.1 Defining the geometry: `metric` objects

EinsteinPy provides a way to define the background geometry, on which the code would deal with the relativistic dynamics. This geometry has a central operating quantity, known as the Metric Tensor, that encapsulates all the geometrical and topological information about the 4D spacetime.

- EinsteinPy provides a `BaseMetric` class, that has various utility functions and a proper template, that can be used to define custom Metric classes. All pre-defined classes in `metric` derive from this class.
- The central quantity required to simulate trajectory of a particle in a gravitational field are the metric derivatives, that can be succinctly written using Christoffel Symbols.
- EinsteinPy provides an easy to use interface to calculate these symbols.

- BaseMetric also provides support for `f_vec` and `perturbation`, where `f_vec` corresponds to the RHS of the geodesic equation and `perturbation` is a linear Kerr-Schild Perturbation, that can be defined on the underlying metric.
- Note that, EinsteinPy does not perform physical checks on `perturbation` currently, and so, users should exercise caution while using it.

We provide an example below, showing how to calculate Time-like Geodesics in Schwarzschild spacetime.

Schwarzschild Metric

EinsteinPy provides an intuitive interface for calculating time-like geodesics in Schwarzschild spacetime.

First of all, we import all the relevant modules and classes:

```
import numpy as np

from einsteinpy.coordinates.utils import four_position, stacked_vec
from einsteinpy.geodesic import Geodesic
from einsteinpy.metric import Schwarzschild
```

Defining initial parameters and our Metric Object

Now, we define the initial parameters, that specify the Schwarzschild metric and our test particle.

```
M = 6e24  # Mass
t = 0.  # Coordinate Time (has no effect in this case, as Schwarzschild␣
↪metric is static)
x_vec = np.array([130.0, np.pi / 2, -np.pi / 8])  # 3-Position of test␣
↪particle
v_vec = np.array([0.0, 0.0, 1900.0])  # 3-Velocity of test particle

ms_cov = Schwarzschild(M=M) # Schwarzschild Metric Object
x_4vec = four_position(t, x_vec) # Getting Position 4-Vector
ms_cov_mat = ms_cov.metric_covariant(x_4vec) # Calculating Schwarzschild␣
↪Metric at x_4vec
init_vec = stacked_vec(ms_cov_mat, t, x_vec, v_vec, time_like=True) #␣
↪Contains 4-Pos and 4-Vel
```

Calculating Trajectory/Time-like Geodesic

After creating the metric object and the initial vector, we can use *Geodesic* to create a Geodesic object, that automatically calculates the trajectory.

```
# Calculating Geodesic
geod = Geodesic(metric=ms_cov, init_vec=init_vec, end_lambda=0.002, step_
↪size=5e-8)
# Getting a descriptive summary on geod
print(geod)
```

```
Geodesic Object:

Metric = ((
```

(continues on next page)

(continued from previous page)

```
Name: (Schwarzschild Metric),
Coordinates: (S),
Mass: (6e+24),
Spin parameter: (0),
Charge: (0),
Schwarzschild Radius: (0.008911392322942397)
)),

Initial Vector = ([ 0.00000000e+00  1.30000000e+02  1.57079633e+00 -3.
↪92699082e-01
1.00003462e+00  0.00000000e+00  0.00000000e+00  1.90000000e+03]),

Trajectory = ([[ 0.00000000e+00  1.20104339e+02 -4.97488462e+01 ...  9.
↪45228078e+04
2.28198245e+05  0.00000000e+00]
[ 4.00013846e-08  1.20108103e+02 -4.97397110e+01 ...  9.36471118e+04
2.28560931e+05 -5.80379473e-14]
[ 4.40015231e-07  1.20143810e+02 -4.96475618e+01 ...  8.48885265e+04
2.32184177e+05 -6.38424865e-13]
...
[ 1.99928576e-03  1.29695466e+02 -6.52793459e-01 ...  1.20900076e+05
2.46971585e+05 -1.86135457e-10]
[ 1.99968577e-03  1.29741922e+02 -5.53995726e-01 ...  1.11380963e+05
2.47015864e+05 -1.74024168e-10]
[ 2.00008578e-03  1.29784572e+02 -4.55181739e-01 ...  1.01868292e+05
2.47052855e+05 -1.61922169e-10]])
```

1.2.2 Bodies Module: `bodies`

EinsteinPy has a module to define the attractor and revolving bodies, using which plotting and geodesic calculation becomes much easier.

Importing all the relevant modules and classes :

```
import numpy as np
from astropy import units as u
from einsteinpy.coordinates import BoyerLindquistDifferential
from einsteinpy.metric import Kerr
from einsteinpy.bodies import Body
from einsteinpy.geodesic import Geodesic
```

Defining various astronomical bodies :

```
spin_factor = 0.3 * u.m
Attractor = Body(name="BH", mass = 1.989e30 * u.kg, a = spin_factor)
BL_obj = BoyerLindquistDifferential(50e5 * u.km, np.pi / 2 * u.rad, np.pi *␣
↪u.rad,
                                    0 * u.km / u.s, 0 * u.rad / u.s, 0 * u.
↪rad / u.s,
                                    spin_factor)
Particle = Body(differential = BL_obj, parent = Attractor)
geodesic = Geodesic(body = Particle, end_lambda = ((1 * u.year).to(u.s)).
↪value / 930,
                    step_size = ((0.02 * u.min).to(u.s)).value,
                    metric=Kerr)
geodesic.trajectory  # get the values of the trajectory
```

Plotting the trajectory :

```
from einsteinpy.plotting import GeodesicPlotter
obj = GeodesicPlotter()
obj.plot(geodesic)
obj.show()
```

1.2.3 Utilities: `utils`

EinsteinPy provides a great set of utility functions which are frequently used in general and numerical relativity.

- Conversion of Coordinates (both position & velocity)
- Cartesian/Spherical
- Cartesian/Boyer-Lindquist
- Calculation of Schwarzschild Geometry related quantities
- Schwarzschild Radius
- Rate of change of coordinate time w.r.t. proper time

Coordinate Conversion

In a short example, we would see coordinate conversion between Cartesian and Boyer-Lindquist Coordinates.

Using the functions:

- `to_cartesian`
- `to_bl`

```
import numpy as np
from astropy import units as u
from einsteinpy.coordinates import BoyerLindquistDifferential,␣
↪CartesianDifferential, Cartesian, BoyerLindquist

a = 0.5 * u.km

pos_vec = Cartesian(.265003774 * u.km, -153.000000e-03 * u.km,  0 * u.km)

bl_pos = pos_vec.to_bl(a)
print(bl_pos)

cartsn_pos = bl_pos.to_cartesian(a)
print(cartsn_pos)

pos_vel_coord = CartesianDifferential(.265003774 * u.km, -153.000000e-03␣
↪* u.km,  0 * u.km,
                          145.45557 * u.km/u.s, 251.93643748389 * u.km/u.
↪s, 0 * u.km/u.s)

bl_coord = pos_vel_coord.bl_differential(a)
bl_coord = bl_coord.si_values()
bl_vel = bl_coord[3:]
print(bl_vel)
```

(continues on next page)

(continued from previous page)

```
cartsn_coord = bl_coord.cartesian_differential(a)
cartsn_coord = cartsn_coord.si_values()
cartsn_vel = cartsn_coord[3:]
print(cartsn_vel)
```

```
[ 200.  -100.    20.5]
[224.54398697   1.47937288  -0.46364761]
```

Symbolic Calculations

EinsteinPy also supports symbolic calculations in `symbolic`

```
import sympy
from einsteinpy.symbolic import SchwarzschildMetric, ChristoffelSymbols

m = SchwarzschildMetric()
ch = ChristoffelSymbols.from_metric(m)
print(ch[1,2,:])
```

```
[0, 0, -r*(-a/r + 1), 0]
```

```
import sympy
from einsteinpy.symbolic import SchwarzschildMetric, EinsteinTensor

m = SchwarzschildMetric()
G1 = EinsteinTensor.from_metric(m)
print(G1.arr)
```

```
[[a*c**2*(-a + r)/r**4 + a*c**2*(a - r)/r**4, 0, 0, 0], [0, a/(r**2*(a - r))␣
↪+ a/(r**2*(-a + r)), 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Future Plans

- Support for null-geodesics in different geometries
- Ultimate goal is providing numerical solutions for Einstein's equations for arbitrarily complex matter distribution.
- Relativistic hydrodynamics

1.3 Vacuum Solutions to Einstein's Field Equations

1.3.1 Einstein's Equation

Einstein's Field Equation(EFE) is a ten component tensor equation which relates local space-time curvature with local energy and momentum. In short, they determine the metric tensor of a spacetime given arrangement of stress-energy in space-time. The EFE is given by

$$R_{\mu\nu} - \frac{1}{2}R\,g_{\mu\nu} + \Lambda g_{\mu\nu} = \frac{8\pi G}{c^4}T_{\mu\nu}$$

Here, $R_{\mu\nu}$ is the Ricci Tensor, R is the curvature scalar(contraction of Ricci Tensor), $g_{\mu\nu}$ is the metric tensor, Λ is the cosmological constant and lastly, $T_{\mu\nu}$ is the stress-energy tensor. All the other variables hold their usual meaning.

1.3.2 Metric Tensor

The metric tensor gives us the differential length element for each durection of space. Small distance in a N-dimensional space is given by :

- $ds^2 = g_{ij}dx_i dx_j$

The tensor is constructed when each g_{ij} is put in it's position in a rank-2 tensor. For example, metric tensor in a spherical coordinate system is given by:

- $g_{00} = 1$
- $g_{11} = r^2$
- $g_{22} = r^2 sin^2\theta$
- $g_{ij} = 0$ when $i \neq j$

We can see the off-diagonal component of the metric to be equal to *0* as it is an orthogonal coordinate system, i.e. all the axis are perpendicular to each other. However it is not always the case. For example, a euclidean space defined by vectors *i*, *j* and *j+k* is a flat space but the metric tensor would surely contain off-diagonal components.

1.3.3 Notion of Curved Space

Imagine a bug travelling across a 2-D paper folded into a cone. The bug can't see up and down, so he lives in a 2d world, but still he can experience the curvature, as after a long journey, he would come back at the position where he started. For him space is not infinite.

Mathematically, curvature of a space is given by Riemann Curvature Tensor, whose contraction is Ricii Tensor, and taking its trace yields a scalar called Ricci Scalar or Curvature Scalar.

Straight lines in Curved Space

Imagine driving a car on a hilly terrain keeping the steering absolutely straight. The trajectory followed by the car, gives us the notion of geodesics. Geodesics are like straight lines in higher dimensional(maybe curved) space.

Mathematically, geodesics are calculated by solving set of differential equation for each space(time) component using the equation:

- $\ddot{x}_i + 0.5 * g^{im} * (\partial_l g_{mk} + \partial_k g_{ml} - \partial_m g_{kl})\dot{x}_k\dot{x}_l = 0$

  which can be re-written as

- $\ddot{x}_i + \Gamma^i_{kl}\dot{x}_k\dot{x}_l = 0$

  where Γ is Christoffel symbol of the second kind.

Christoffel symbols can be encapsulated in a rank-3 tensor which is symmetric over it's lower indices. Coming back to Riemann Curvature Tensor, which is derived from Christoffel symbols using the equation

- $R^i_{abc} = \partial_b \Gamma^i_{ca} - \partial_c \Gamma^i_{ba} + \Gamma^i_{bm}\Gamma^m_{ca} - \Gamma^i_{cm}\Gamma^m_{ba}$

Of course, Einstein's indicial notation applies everywhere.

Contraction of Riemann Tensor gives us Ricci Tensor, on which taking trace gives Ricci or Curvature scalar. A space with no curvature has Riemann Tensor as zero.

1.3.4 Exact Solutions of EFE

Schwarzschild Metric

It is the first exact solution of EFE given by Karl Schwarzschild, for a limited case of single spherical non-rotating mass. The metric is given as:

- $d\tau^2 = -(1 - r_s/r)dt^2 + (1 - r_s/r)^{-1}dr^2 + r^2 d\theta^2/c^2 + r^2 sin^2\theta d\phi^2/c^2$

  where $r_s = 2 * G * M/c^2$

and is called the Schwarzschild Radius, a point beyond where space and time flips and any object inside the radius would require speed greater than speed of light to escape singularity, where the curvature of space becomes infinite and so is the case with the tidal forces. Putting $r = \infty$, we see that the metric transforms to a metric for a flat space defined by spherical coordinates.

τ is the proper time, the time experienced by the particle in motion in the space-time while t is the coordinate time observed by an observer at infinity.

Using the metric in the above discussed geodesic equation gives the four-position and four-velocity of a particle for a given range of τ. The differential equations can be solved by supplying the initial positions and velocities.

Kerr Metric and Kerr-Newman Metric

Kerr-Newman metric is also an exact solution of EFE. It deals with spinning, charged massive body as the solution has axial symettry. A quick search on google would give the exact metric as it is quite exhaustive.

Kerr-Newman metric is the most general vacuum solution consisting of a single body at the center.

Kerr metric is a specific case of Kerr-Newman where charge on the body $Q = 0$. Schwarzschild metric can be derived from Kerr-Newman solution by putting charge and spin as zero $Q = 0$, $a = 0$.

1.4 Jupyter notebooks

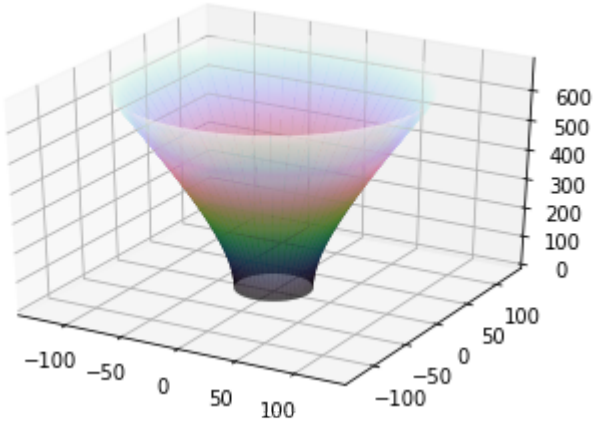
1.4.1 Plotting Spacial Hypersurface Embedding for Schwarzschild Space-Time

[1]:
```
from einsteinpy.hypersurface import SchwarzschildEmbedding
from einsteinpy.plotting import HypersurfacePlotter
from astropy import units as u
```

Declaring embedding object with specified mass of the body and plotting the embedding hypersurface for Schwarzschild spacetime

[2]:
```
surface_obj = SchwarzschildEmbedding(5.927e23 * u.kg)
```

[3]:
```
surface = HypersurfacePlotter(embedding=surface_obj, plot_type='surface')
surface.plot()
surface.show()
```



The plotted embedding has initial Schwarzschild radial coordinate to be greater than schwarzschild radius but the embedding can be defined for coordinates greater than 9m/4. The Schwarzschild spacetime is a static spacetime and thus the embeddings can be obtained by considering fermat's surfaces of stationary time coordinate and thus this surface also represent the spacial geometry on which light rays would trace their paths along geodesics of this surface (spacially)!

1.4.2 Shadow cast by an thin emission disk around a Schwarzschild Black Hole

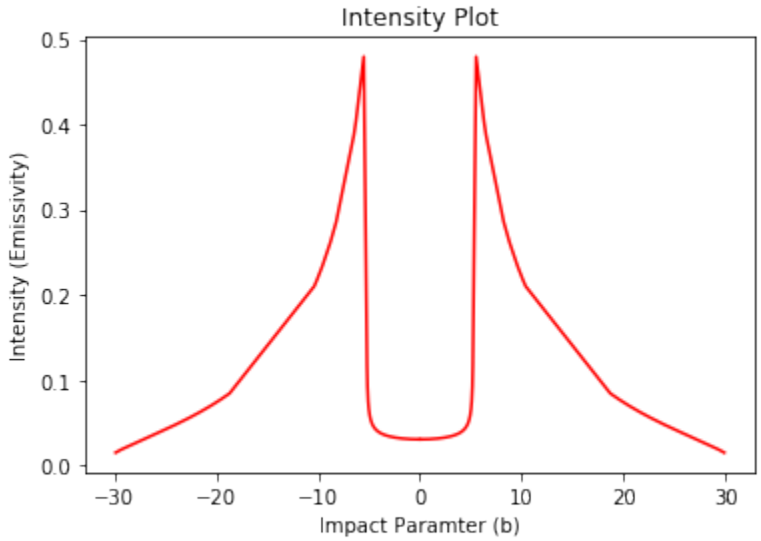
[1]:
```
import astropy.units as u

from einsteinpy.rays import Shadow
from einsteinpy.plotting import ShadowPlotter
```

Defining the conditions

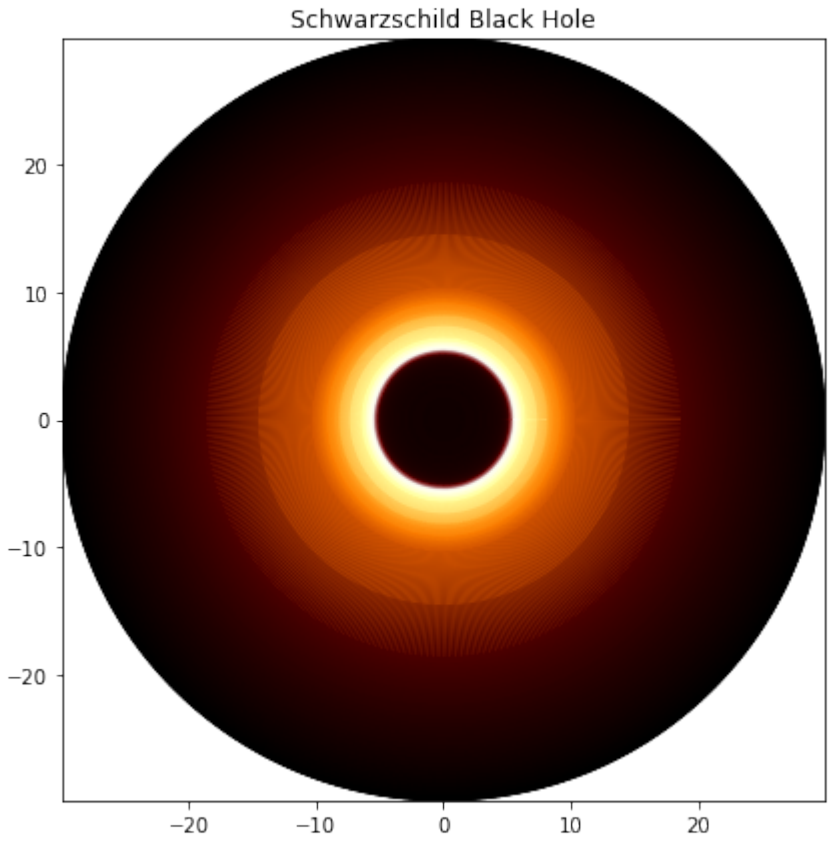
[2]:
```
mass = 1 * u.kg
fov = 30 * u.km
# What field of view is the user expecting
```

[3]:
```
shadow = Shadow(mass=mass, fov=fov, n_rays=1000)
```

[4]:
```
obj = ShadowPlotter(shadow=shadow, is_line_plot=True)
obj.plot()
obj.show()
```



[5]:
```
obj = ShadowPlotter(shadow=shadow, is_line_plot=False)
obj.plot()
obj.show()
```



1.4.3 Animations in EinsteinPy

Importing required modules

[1]:
```
import numpy as np

from einsteinpy.geodesic import Timelike
from einsteinpy.plotting import StaticGeodesicPlotter
```

Defining various parameters

- Initial position & momentum of the test partcle
- Spin of the Kerr Black Hole
- Other solver parameters

Note that, we are working in M-Units ($G = c = M = 1$).

[2]:
```
# Constant Radius Orbit
position = [4, np.pi / 3, 0.]
momentum = [0., 0.767851, 2.]
a = 0.99
end_lambda = 200.
step_size = 0.5
```

Calculating geodesic, using the Julia back-end

[3]:
```
geod = Timelike(
    position=position,
    momentum=momentum,
    a=a,
    end_lambda=end_lambda,
    step_size=step_size,
    return_cartesian=True,
    julia=True
)
```

Animating

[5]:
```
%matplotlib nbagg
sgpl = StaticGeodesicPlotter()
sgpl.animate(geod, interval=1)
sgpl.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Saving animation as .gif

[6]:
```
sgpl.ani.save('animation.gif', writer='imagemagick', fps=60)
```

1.4.4 Using Geodesics (Back-ends & Plotting)

Importing required modules

Note that, for the Julia backend to work, you should have ``einsteinpy_geodesics`` installed, along with its dependencies (e.g. ``julia``). Read more `here <https://github.com/einsteinpy/einsteinpy-geodesics/#requirements>`__.

[1]:
```
import numpy as np

from einsteinpy.geodesic import Geodesic, Timelike, Nulllike
from einsteinpy.plotting import GeodesicPlotter, StaticGeodesicPlotter,␣
↪InteractiveGeodesicPlotter
```

Example 1: Exploring Schwarzschild Time-like Spiral Capture, using Python Backend and GeodesicPlotter

Defining initial conditions

[2]:
```
# Initial Conditions
position = [2.15, np.pi / 2, 0.]
momentum = [0., 0., -1.5]
a = 0. # Schwarzschild Black Hole
end_lambda = 10.
step_size = 0.005
return_cartesian = True
time_like = True
julia = False # Using Python
```

Calculating Geodesic

[3]:
```
geod = Geodesic(
    position=position,
    momentum=momentum,
    a=a,
    end_lambda=end_lambda,
    step_size=step_size,
    time_like=time_like, # Necessary to switch between Time-like and Null-like␣
↪Geodesics, while using `Geodesic`
    return_cartesian=return_cartesian,
    julia=julia
)

geod
```

```
e:\coding\gsoc\github repos\myfork\einsteinpy\src\einsteinpy\geodesic\geodesic.py:136:
↪ RuntimeWarning:


                Using Python backend to solve the system. This backend is currently␣
↪in beta and the
                solver may not be stable for certain sets of conditions, e.g. long␣
↪simulations
                (`end_lambda > 50.`) or high initial radial distances (`position[0] >␣
↪~5.`).
                In these cases or if the output does not seem accurate, it is highly␣
↪recommended to
                switch to the Julia backend, by setting `julia=True`, in the␣
↪constructor call.
```

[3]:
```
Geodesic Object:
            Type = (Time-like),
            Position = ([2.15, 1.5707963267948966, 0.0]),
            Momentum = ([0.0, 0.0, -1.5]),
            Spin Parameter = (0.0)
            Solver details = (
                Backend = (Python)
                Step-size = (0.005),
                End-Lambda = (10.0)
                Trajectory = (
                    (array([0.000e+00, 5.000e-03, 1.000e-02, ..., 9.990e+00, 9.
↪995e+00,
       1.000e+01]), array([[ 2.15000000e+00,  0.00000000e+00,  1.31649531e-16,
         0.00000000e+00,  0.00000000e+00, -1.50000000e+00],
       [ 2.14999615e+00, -1.61252735e-02,  1.31652998e-16,
         2.26452642e-02,  1.49020159e-19, -1.50000000e+00],
       [ 2.14998455e+00, -3.22521873e-02,  1.31663397e-16,
         4.52748906e-02,  2.98024624e-19, -1.50000000e+00],
       ...,
       [-1.08571331e+01, -9.57951275e+00,  8.86589316e-16,
         1.14573395e+00,  4.59929900e-17, -1.50000000e+00],
       [-1.09329973e+01, -9.50157334e+00,  8.86940089e-16,
         1.14568933e+00,  4.59962746e-17, -1.50000000e+00],
       [-1.10083027e+01, -9.42303446e+00,  8.87290849e-16,
         1.14564476e+00,  4.59995565e-17, -1.50000000e+00]]))
                ),
                Output Position Coordinate System = (Cartesian)
            )
```

Plotting using `GeodesicPlotter`

Note that, `GeodesicPlotter` automatically switches between "Static" and "Interactive" plots. Since, we are in a Jupyter Notebook or Interactive Environment, it uses the "Interactive" backend.

[4]:
```
gpl = GeodesicPlotter()
```

[5]:
```
gpl.plot(geod, color="green")
gpl.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[6]:
```
gpl.clear() # In Interactive mode, `clear()` must be called before drawing another␣
↪plot, to avoid overlap
gpl.plot2D(geod, coordinates=(1, 2), color="green")
gpl.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[7]:
```
gpl.clear()
gpl.plot2D(geod, coordinates=(1, 3), color="green")
gpl.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Clearly, the geodesic is equatorial, which is as expected.

Now, let us visualize the variation of coordinates with respect to the affine parameter:

[8]:
```
gpl.clear()
gpl.parametric_plot(geod, colors=("red", "green", "blue"))
gpl.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Example 2 - Exploring Kerr Extremal Time-like Constant Radius Orbit, using Julia Backend and `StaticGeodesicPlotter`

Defining initial conditions

[9]:
```
# Initial Conditions
position = [4, np.pi / 3, 0.]
momentum = [0., 0.767851, 2.]
a = 0.99 # Extremal Kerr Black Hole
end_lambda = 300.
step_size = 1.
return_cartesian = True
julia = True # Using Julia
```

Calculating Geodesic

[10]:
```
geod = Timelike(
    position=position,
    momentum=momentum,
    a=a,
    end_lambda=end_lambda,
    step_size=step_size,
    return_cartesian=return_cartesian,
    julia=julia
)

geod
```

[10]:
```
Geodesic Object:
            Type = (Time-like),
            Position = ([4, 1.0471975511965976, 0.0]),
            Momentum = ([0.0, 0.767851, 2.0]),
            Spin Parameter = (0.99)
            Solver details = (
                Backend = (Julia)
                Step-size = (1.0),
                End-Lambda = (300.0)
                Trajectory = (
                    (array([  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.
↪,  10.,
        11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,  20.,  21.,
        22.,  23.,  24.,  25.,  26.,  27.,  28.,  29.,  30.,  31.,  32.,
        33.,  34.,  35.,  36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,
        44.,  45.,  46.,  47.,  48.,  49.,  50.,  51.,  52.,  53.,  54.,
        55.,  56.,  57.,  58.,  59.,  60.,  61.,  62.,  63.,  64.,  65.,
        66.,  67.,  68.,  69.,  70.,  71.,  72.,  73.,  74.,  75.,  76.,
        77.,  78.,  79.,  80.,  81.,  82.,  83.,  84.,  85.,  86.,  87.,
        88.,  89.,  90.,  91.,  92.,  93.,  94.,  95.,  96.,  97.,  98.,
        99., 100., 101., 102., 103., 104., 105., 106., 107., 108., 109.,
       110., 111., 112., 113., 114., 115., 116., 117., 118., 119., 120.,
       121., 122., 123., 124., 125., 126., 127., 128., 129., 130., 131.,
       132., 133., 134., 135., 136., 137., 138., 139., 140., 141., 142.,
       143., 144., 145., 146., 147., 148., 149., 150., 151., 152., 153.,
       154., 155., 156., 157., 158., 159., 160., 161., 162., 163., 164.,
       165., 166., 167., 168., 169., 170., 171., 172., 173., 174., 175.,
       176., 177., 178., 179., 180., 181., 182., 183., 184., 185., 186.,
       187., 188., 189., 190., 191., 192., 193., 194., 195., 196., 197.,
       198., 199., 200., 201., 202., 203., 204., 205., 206., 207., 208.,
       209., 210., 211., 212., 213., 214., 215., 216., 217., 218., 219.,
       220., 221., 222., 223., 224., 225., 226., 227., 228., 229., 230.,
       231., 232., 233., 234., 235., 236., 237., 238., 239., 240., 241.,
       242., 243., 244., 245., 246., 247., 248., 249., 250., 251., 252.,
       253., 254., 255., 256., 257., 258., 259., 260., 261., 262., 263.,
       264., 265., 266., 267., 268., 269., 270., 271., 272., 273., 274.,
       275., 276., 277., 278., 279., 280., 281., 282., 283., 284., 285.,
       286., 287., 288., 289., 290., 291., 292., 293., 294., 295., 296.,
       297., 298., 299., 300.]), array([[ 3.46410162e+00,  0.00000000e+00,  2.
↪00000000e+00,
         0.00000000e+00,  7.67851000e-01,  2.00000000e+00],
       [ 3.31204818e+00, -6.75743266e-01,  2.13983951e+00,
        -2.35680608e-03,  5.57091347e-01,  2.00000000e+00],
       [ 3.04768664e+00, -1.32417637e+00,  2.23112216e+00,
```

(continues on next page)

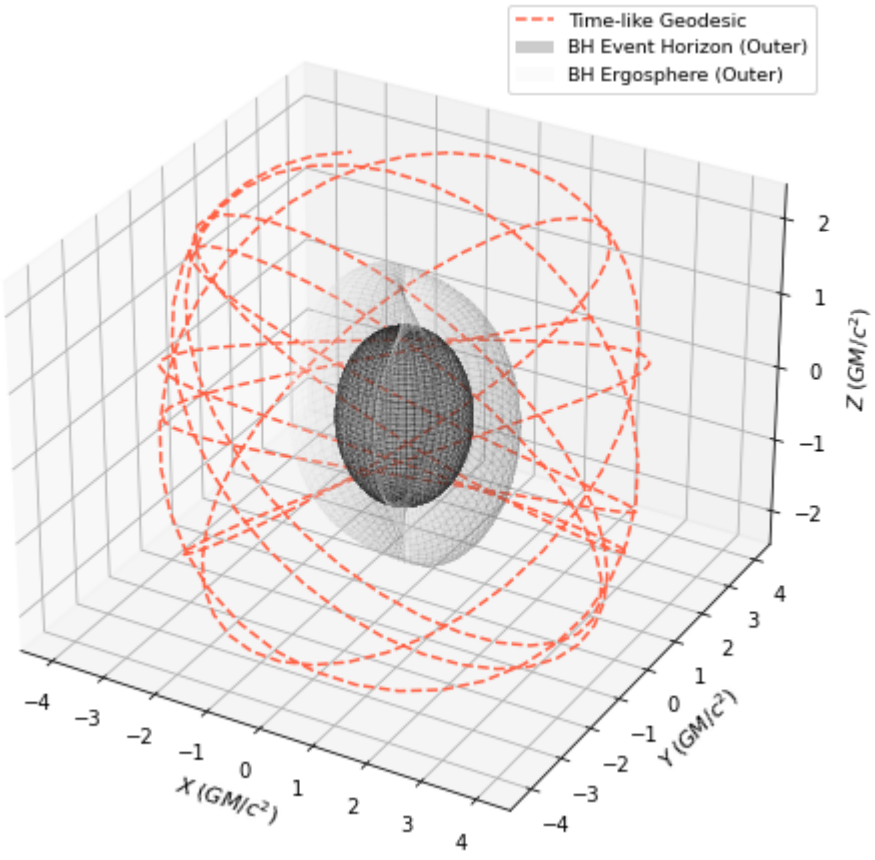
(continued from previous page)

```
        -4.23238896e-03,  3.20708859e-01,  2.00000000e+00],
       ...,
       [-3.11522440e+00,  1.20649964e+00,  2.34482383e+00,
        -6.17729017e-03,  1.41991524e-01,  2.00000000e+00],
       [-2.79842823e+00,  1.82638208e+00,  2.34970281e+00,
        -6.77531224e-03, -1.13632352e-01,  2.00000000e+00],
       [-2.38699763e+00,  2.38923406e+00,  2.30474976e+00,
        -6.41027572e-03, -3.63135128e-01,  2.00000000e+00]]))
               ),
               Output Position Coordinate System = (Cartesian)
           )
```

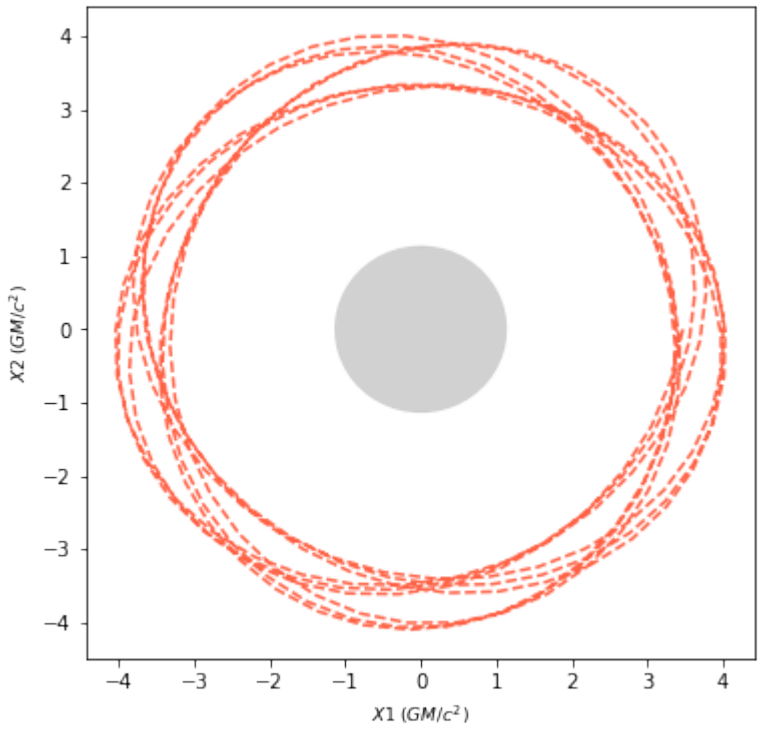
Plotting using `StaticGeodesicPlotter`

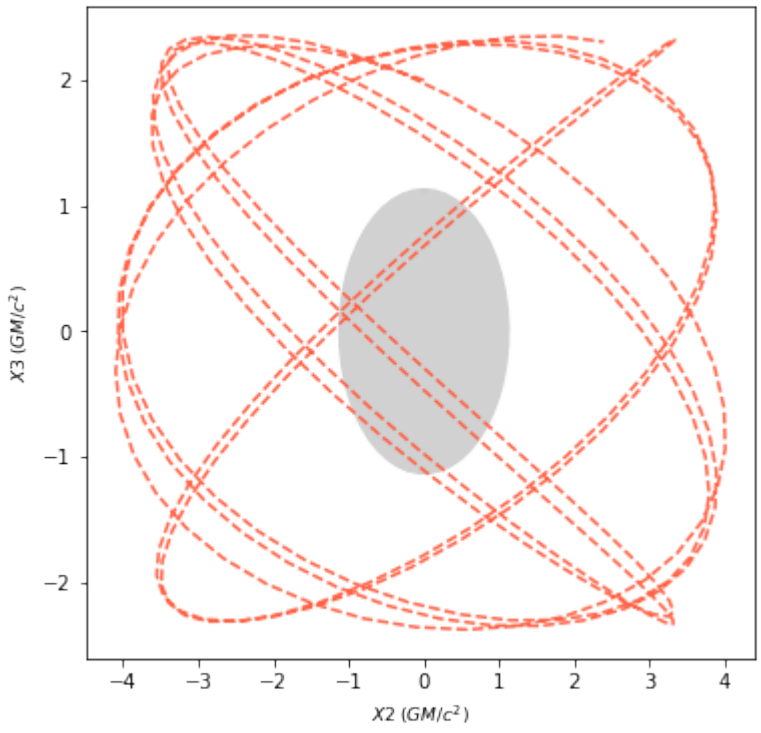
[11]:
```
sgpl = StaticGeodesicPlotter(
    bh_colors=("black", "white"), # Colors for BH surfaces, Event Horizon & Ergosphere
    draw_ergosphere=True # Ergosphere will be drawn
)
```

[12]:
```
sgpl.plot(geod, color="tomato", figsize=(8, 8)) # figsize is in inches
sgpl.show()
```



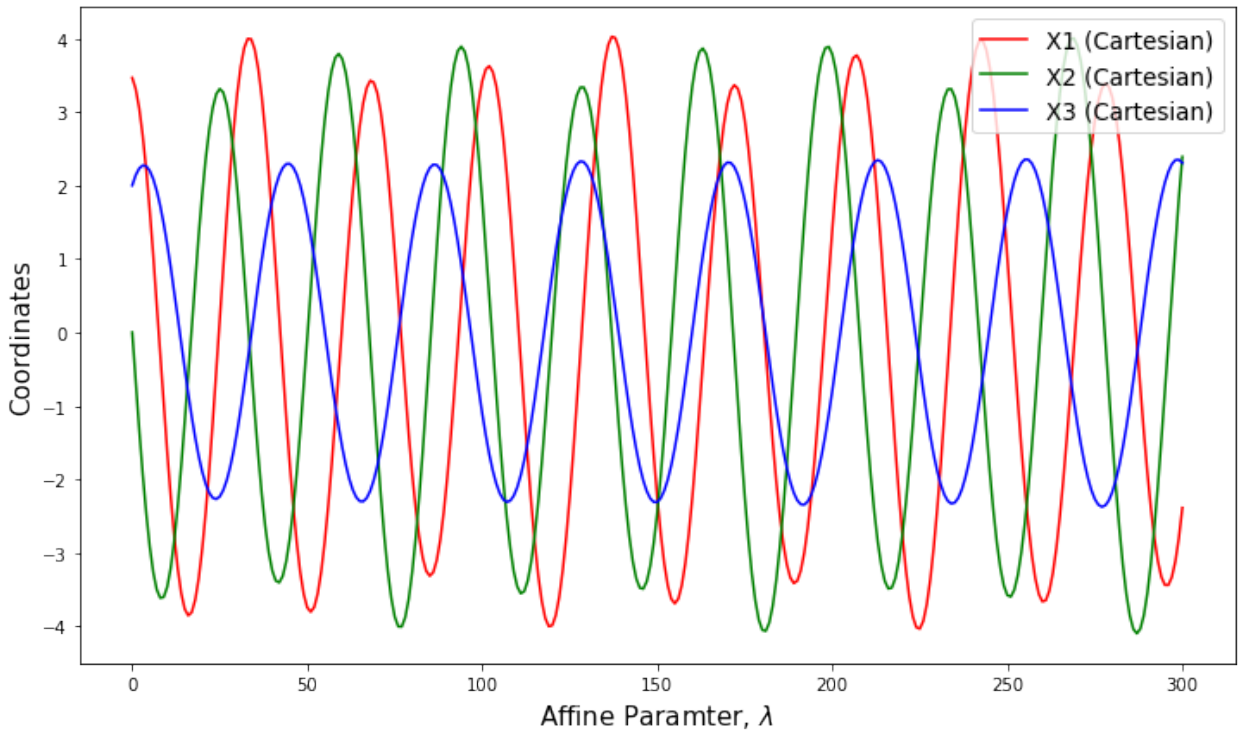
[13]:
```
sgpl.plot2D(geod, coordinates=(1, 2), figsize=(6, 6), color="tomato") # X vs Y
sgpl.plot2D(geod, coordinates=(2, 3), figsize=(6, 6), color="tomato") # Y vs Z
```





Let us explore this plot parametrically and visualize the coordinate variations, with respect to the affine parameter.

[14]:
```
sgpl.parametric_plot(geod, figsize=(12, 7), colors=("red", "green", "blue"))
sgpl.show()
```



We can also produce high-quality animations with `animate`, as shown below.

[15]:
```
# Using nbagg matplotlib Jupyter back-end for Interactive Matplotlib Plots - Necessary for animations
%matplotlib nbagg
sgpl.animate(geod, interval=10, color="tomato")
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[16]:
```
# Saving the animation to file
sgpl.ani.save("CoolKerrGeodesic.gif", write="imagemagick", fps=60) # You should have "imagemagick" isntalled in your system
```

Example 3 - Kerr Null-like Capture (Frame Dragging), using Julia Backend and InteractiveGeodesicPlotter

Defining initial conditions

[17]:
```
# Initial Conditions
position = [2.5, np.pi / 2, 0.]
momentum = [0., 0., -2.]
a = 0.99 # Extremal Kerr Black Hole
end_lambda = 150.
```

(continues on next page)

(continued from previous page)

```
step_size = 0.0005 # Low step_size is required for good approximation in a strong␣
↪gravity region
return_cartesian = True
julia = True # Using Julia
```

Calculating Geodesic

[18]:
```
geod = Nulllike(
    position=position,
    momentum=momentum,
    a=a,
    end_lambda=end_lambda,
    step_size=step_size,
    return_cartesian=return_cartesian,
    julia=julia
)

geod
```

```
e:\coding\winpython\wpy64-3740\python-3.7.4.amd64\lib\site-packages\einsteinpy_
↪geodesics\geodesics_wrapper.py:84: RuntimeWarning:

Test particle has reached the Event Horizon.
```

[18]:
```
Geodesic Object:
            Type = (Null-like),
            Position = ([2.5, 1.5707963267948966, 0.0]),
            Momentum = ([0.0, 0.0, -2.0]),
            Spin Parameter = (0.99)
            Solver details = (
                Backend = (Julia)
                Step-size = (0.0005),
                End-Lambda = (150.0)
                Trajectory = (
                    (array([0.00000000e+00, 5.00000000e-04, 1.00000000e-03, ...,
       3.71700000e+00, 3.71749241e+00, 3.71749241e+00]), array([[ 2.50000000e+00,  0.
↪00000000e+00,  1.53080850e-16,
         0.00000000e+00,  0.00000000e+00, -2.00000000e+00],
       [ 2.49999998e+00,  1.46505775e-04,  1.53080849e-16,
         1.52261067e-04, -1.94472683e-20, -2.00000000e+00],
       [ 2.49999993e+00,  2.93011541e-04,  1.53080847e-16,
         3.04522151e-04, -3.88945371e-20, -2.00000000e+00],
       ...,
       [-1.15166415e+00,  6.23772183e-02,  7.06224520e-17,
         6.40315667e+02, -2.53361259e-16, -2.00000000e+00],
       [-1.10116520e+00,  3.40060011e-01,  7.05689267e-17,
         6.90769982e+02, -1.13876058e-05, -1.99992945e+00],
       [-1.10116520e+00,  3.40060011e-01,  7.05689267e-17,
         6.90769982e+02, -1.13876058e-05, -1.99992945e+00]]))
                ),
                Output Position Coordinate System = (Cartesian)
            )
```

Plotting using `InteractiveGeodesicPlotter`

[19]:
```
igpl = InteractiveGeodesicPlotter(
    bh_colors=("black", "white"), # Colors for BH surfaces, Event Horizon & Ergosphere
    draw_ergosphere=True # Ergosphere will be drawn
)
```

[20]:
```
igpl.plot(geod, color="indigo")
igpl.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[21]:
```
igpl.clear()
igpl.plot2D(geod, coordinates=(1, 2), color="indigo") # X vs Y
igpl.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[22]:
```
igpl.clear()
igpl.plot2D(geod, coordinates=(1, 3), color="indigo") # X vs Z
igpl.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

From the second plot, we note, that the test particle never leaves the equatorial plane, which is as expected.

As before, we can plot the geodesic, parametrically:

[23]:
```
igpl.clear()
igpl.parametric_plot(geod, colors=("red", "green", "blue"))
igpl.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

We can clearly see the exact "moment" the test particle hits the singularity or the Event Horizon.

1.4.5 Visualizing Event Horizon and Ergosphere (Singularities) of Kerr Metric or Black Hole

Importing required modules

[1]:
```
import astropy.units as u
import numpy as np
import astropy.units as u
import matplotlib.pyplot as plt

from einsteinpy.coordinates import BoyerLindquistDifferential
from einsteinpy.metric import Kerr
```

Defining black holes and obtaining singularities

[2]:
```
# Metric or Black Hole parameters - Mass, M and Spin Parameter, a
M = 4e30 * u.kg
a1 = 0.4 * u.one
a2 = 0.9 * u.one # Extremal Kerr Black Hole

# Coordinate object to initialize metric with
# Note that, for this example
# the coordinate values below are irrelevant
bl = BoyerLindquistDifferential(
    t=0. * u.s,
    r=1e3 * u.m,
    theta=np.pi / 2 * u.rad,
    phi=np.pi * u.rad,
    v_r=0. * u.m / u.s,
    v_th=0. * u.rad / u.s,
    v_p=0. * u.rad / u.s,
)

# Defining two Kerr Black Holes, one with a higher spin parameter
kerr1 = Kerr(coords=bl, M=M, a=a1)
kerr2 = Kerr(coords=bl, M=M, a=a2)

# Getting the list of singularities
sing_dict1 = kerr1.singularities()
sing_dict2 = kerr2.singularities()

# Let's check the contents of the dicts
# 'ergosphere' entries should be functions
print(sing_dict1, sing_dict2, sep="\n\n")
```

```
{'inner_ergosphere': <function BaseMetric.singularities.<locals>._in_ergo at␣
↪0x000002AAAB94D438>, 'inner_horizon': 247.98878315867296, 'outer_horizon': 5692.
↪939432136259, 'outer_ergosphere': <function BaseMetric.singularities.<locals>._out_
↪ergo at 0x000002AAAB94DCA8>}

{'inner_ergosphere': <function BaseMetric.singularities.<locals>._in_ergo at␣
↪0x000002AAAA4D5A68>, 'inner_horizon': 1675.668821582463, 'outer_horizon': 4265.
↪259393712469, 'outer_ergosphere': <function BaseMetric.singularities.<locals>._out_
↪ergo at 0x000002AAAB952558>}
```

Preparing singularities for plotting

[3]:
```
# Sampling Polar Angle for plotting in Polar Coordinates
theta = np.linspace(0, 2 * np.pi, 100)

# Ergospheres
# These are functions
Ei1, Eo1 = sing_dict1["inner_ergosphere"], sing_dict1["outer_ergosphere"]
Ei2, Eo2 = sing_dict2["inner_ergosphere"], sing_dict2["outer_ergosphere"]

# Creating lists of points on Ergospheres for different polar angles, for both black␣
↪holes
Ei1_list, Eo1_list = Ei1(theta), Eo1(theta)
Ei2_list, Eo2_list = Ei2(theta), Eo2(theta)
```

(continues on next page)

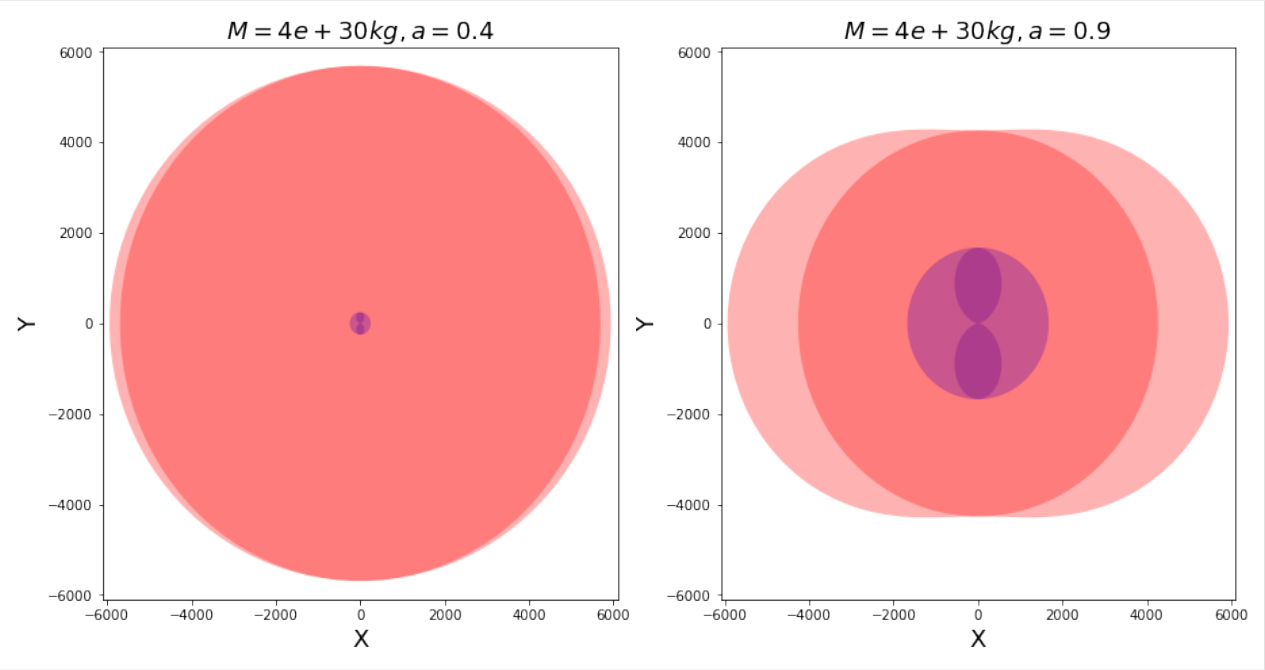
(continued from previous page)

```
# For Black Hole 1 (a = 0.4)
Xei1 = Ei1_list * np.sin(theta)
Yei1 = Ei1_list * np.cos(theta)

Xeo1 = Eo1_list * np.sin(theta)
Yeo1 = Eo1_list * np.cos(theta)

# For Black Hole 2 (a = 0.9)
Xei2 = Ei2_list * np.sin(theta)
Yei2 = Ei2_list * np.cos(theta)

Xeo2 = Eo2_list * np.sin(theta)
Yeo2 = Eo2_list * np.cos(theta)

# Event Horizons
Hi1, Ho1 = sing_dict1["inner_horizon"], sing_dict1["outer_horizon"]
Hi2, Ho2 = sing_dict2["inner_horizon"], sing_dict2["outer_horizon"]

# For Black Hole 1 (a = 0.4)
Xhi1 = Hi1 * np.sin(theta)
Yhi1 = Hi1 * np.cos(theta)

Xho1 = Ho1 * np.sin(theta)
Yho1 = Ho1 * np.cos(theta)

# For Black Hole 2 (a = 0.9)
Xhi2 = Hi2 * np.sin(theta)
Yhi2 = Hi2 * np.cos(theta)

Xho2 = Ho2 * np.sin(theta)
Yho2 = Ho2 * np.cos(theta)
```

Plotting both black holes

[4]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15,7.5))

ax1.fill(Xei1, Yei1, 'b', Xeo1, Yeo1, 'r', Xhi1, Yhi1, 'b', Xho1, Yho1, 'r', alpha=0.
↪3)
ax1.set_title(f"$M = {M}, a = {a1}$", fontsize=18)
ax1.set_xlabel("X", fontsize=18)
ax1.set_ylabel("Y", fontsize=18)
ax1.set_xlim([-6100, 6100])
ax1.set_ylim([-6100, 6100])

ax2.fill(Xei2, Yei2, 'b', Xeo2, Yeo2, 'r', Xhi2, Yhi2, 'b', Xho2, Yho2, 'r', alpha=0.
↪3)
ax2.set_title(f"$M = {M}, a = {a2}$", fontsize=18)
ax2.set_xlabel("X", fontsize=18)
ax2.set_ylabel("Y", fontsize=18)
ax2.set_xlim([-6100, 6100])
ax2.set_ylim([-6100, 6100])
```

[4]:
```
(-6100.0, 6100.0)
```



- The surfaces are clearly visible in the plots. Going radially inward, we have Outer Ergosphere, Outer Event Horizon, Inner Event Horizon and Inner Ergosphere. We can also observe the following:
  - As $a \to 1$ (its maximum attainable value), the individual singularities become prominent.
  - As $a \to 0$, some singularities appear to fade away, leaving us with a single surface, that is the Event Horizon of a Schwarzschild black hole.

1.4.6 Visualizing Frame Dragging in Kerr Spacetime

Importing required modules

[1]:
```
import numpy as np

from einsteinpy.geodesic import Nulllike
from einsteinpy.plotting import StaticGeodesicPlotter
```

Setting up the system

- Initial position & momentum of the test partcle
- Spin of the Kerr Black Hole
- Other solver parameters

Note that, we are working in M-Units ($G = c = M = 1$). Also, setting momentum's ϕ-component to negative, implies an initial retrograde trajectory.

[2]:
```
position = [2.5, np.pi / 2, 0.]
momentum = [0., 0., -2.]
a = 0.99
```

(continues on next page)

(continued from previous page)

```
end_lambda = 150.
step_size = 0.0005
```

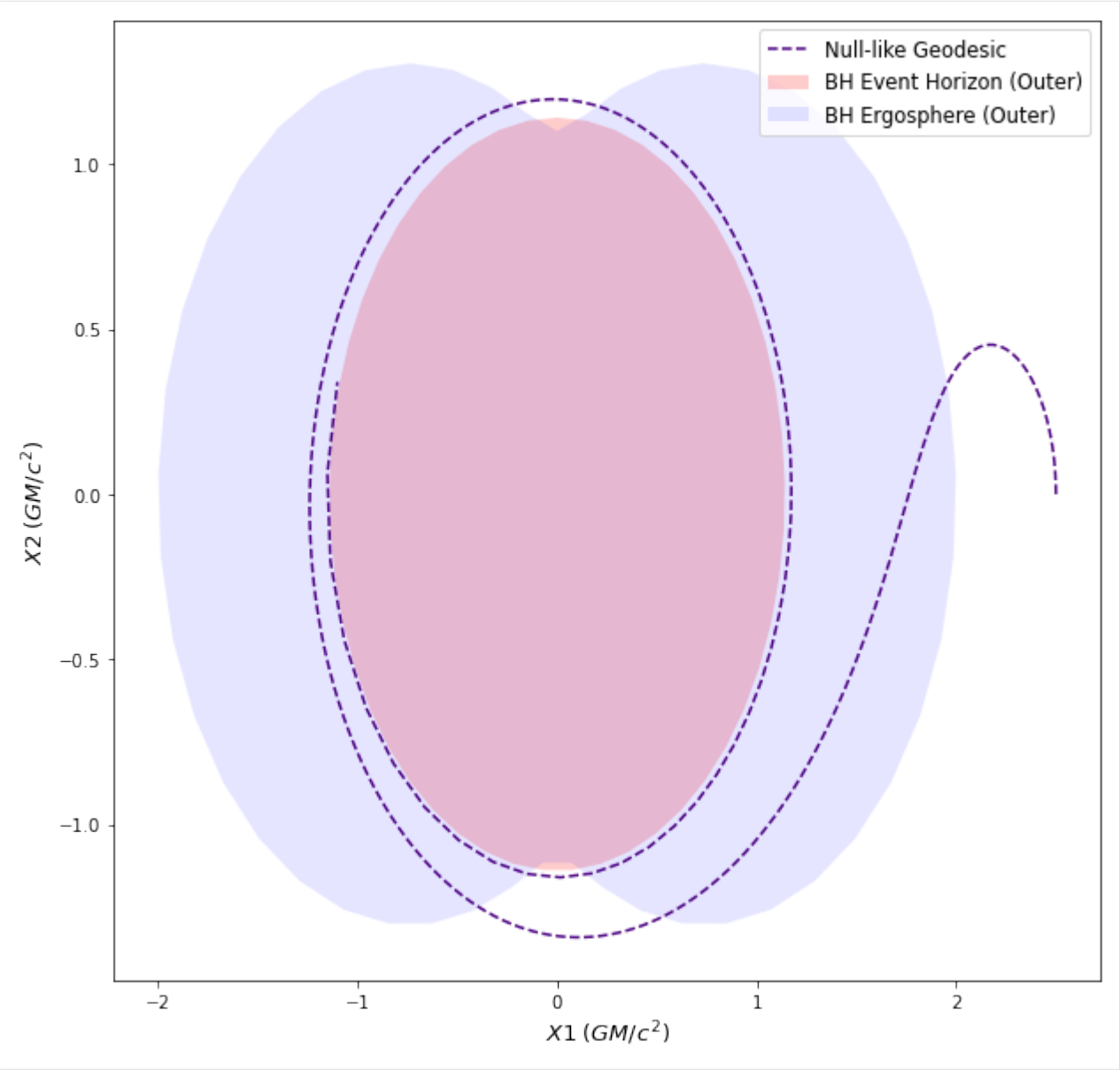
Calculating the geodesic, using the Julia back-end

[3]:
```
geod = Nulllike(
    position=position,
    momentum=momentum,
    a=a,
    end_lambda=end_lambda,
    step_size=step_size,
    return_cartesian=True,
    julia=True
)
```

```
e:\coding\gsoc\github repos\myfork\einsteinpy\src\einsteinpy\geodesic\utils.py:307:␣
↪RuntimeWarning:

Test particle has reached the Event Horizon.
```

Plotting the geodesic in 2D

[4]:
```
sgpl = StaticGeodesicPlotter(bh_colors=("red", "blue"))
sgpl.plot2D(geod, coordinates=(1, 2), figsize=(10, 10), color="indigo") # Plot X vs Y
sgpl.show()
```



As can be seen in the plot above, the photon's trajectory is reversed, due to frame-dragging effects, so that, it moves in the direction of the black hole's spin, before eventually falling into the black hole.

1.4.7 Visualizing Precession in Schwarzschild Spacetime

Importing required modules

[1]:
```
import numpy as np

from einsteinpy.geodesic import Timelike
from einsteinpy.plotting.geodesic import GeodesicPlotter
```

Setting up the system

- Initial position & momentum of the test partcle
- Spin of the Schwarzschild Black Hole (= 0)
- Other solver parameters

Note that, we are working in M-Units ($G = c = M = 1$). Also, since the Schwarzschild spacetime has spherical symmetry, the values of the angular components do not affect the end result (We can always rotate our coordinate system to bring the geodesic in the equatorial plane). Hence, we set $\theta = \pi/2$ (equatorial plane), with initial $p_\theta = 0$, which implies, that the geodesic should stay in the equatorial plane.

[2]:
```
position = [40, np.pi / 2, 0.]
momentum = [0., 0., 4.]
a = 0.
end_lambda = 20000.
step_size = 0.5
```

Calculating the geodesic, using the Julia back-end

[3]:
```
geod = Timelike(
    position=position,
    momentum=momentum,
    a=a,
    end_lambda=end_lambda,
    step_size=step_size,
    return_cartesian=True,
    julia=True
)
```

Plotting the geodesic

[4]:
```
gpl = GeodesicPlotter()
gpl.plot(geod)
gpl.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Apsidal Precession is easily observed through the plot, above, and as expected, the geodesic is confined to the equatorial plane. We can visualize this better, with a few 2D plots.

[5]:
```
gpl.clear()
gpl.plot2D(geod, coordinates=(1, 2)) # Plot X & Y
gpl.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[6]:
```
gpl.clear()
gpl.plot2D(geod, coordinates=(1, 3)) # Plot X & Z
gpl.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[7]:
```
gpl.clear()
gpl.plot2D(geod, coordinates=(2, 3)) # Plot Y & Z
gpl.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Let's see, how the coordinates change, with λ (Affine Parameter).

[8]:
```
gpl.clear()
gpl.parametric_plot(geod, colors=("cyan", "magenta", "lime")) # Plot X, Y, Z vs␣
↪Lambda (Affine Parameter)
gpl.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

The constant lag between $X1$ (x-component) and $X2$ (y-component), in the parametric plot, reaffirms the results from above. Also, $X3 \approx 0$ (z-component) in this plot, which is expected, as the geodesic never leaves the equatorial plane.

1.4.8 Einstein Tensor calculations using Symbolic module

[1]:
```
import numpy as np
import pytest
import sympy
from sympy import cos, simplify, sin, sinh, tensorcontraction
from einsteinpy.symbolic import EinsteinTensor, MetricTensor, RicciScalar

sympy.init_printing()
```

Defining the Anti-de Sitter spacetime Metric

[2]:
```
syms = sympy.symbols("t chi theta phi")
t, ch, th, ph = syms
m = sympy.diag(-1, cos(t) ** 2, cos(t) ** 2 * sinh(ch) ** 2, cos(t) ** 2 * sinh(ch)␣
↪** 2 * sin(th) ** 2).tolist()
metric = MetricTensor(m, syms)
```

Calculating the Einstein Tensor (with both indices covariant)

[3]:
```
einst = EinsteinTensor.from_metric(metric)
einst.tensor()
```

[3]:
$$\begin{bmatrix} \frac{0.5\left(\left(\sin^2(t)-1\right)\sinh^2(\chi)-2\cos^2(t)\sinh^2(\chi)\right)}{\cos^2(t)\sinh^2(\chi)} + \frac{0.5\left(\left(\sin^2(t)-1\right)\sin^2(\theta)\sinh^2(\chi)-2\sin^2(\theta)\cos^2(t)\sinh^2(\chi)\right)}{\sin^2(\theta)\cos^2(t)\sinh^2(\chi)} & 0 & 0 & -0.5\left(\frac{\left(\sin^2(t)-1\right)\sinh^2(\chi)-2\cos^2\right. \\ 0 & & & \\ 0 & & & \\ 0 & & & \end{bmatrix}$$

1.4.9 Lambdify in symbolic module

Importing required modules

[1]:
```
import sympy
from sympy.abc import x, y
from sympy import symbols
from einsteinpy.symbolic import BaseRelativityTensor

sympy.init_printing()
```

Calculating a Base Relativity Tensor

[2]:
```
syms = symbols("x y")
x, y = syms
T = BaseRelativityTensor([[x, 1],[0, x+y]], syms, config="ll")
```

Calling the lambdify function

[3]:
```
args, func = T.tensor_lambdify()
args
```

[3]:
(x, y)

`args` indicates the order in which arguments should be passed to the returned function `func`

Executing the returned function for some value

[4]:
```
func(2, 1)
```

[4]:
$[[2, 1], [0, 3]]$

1.4.10 Contravariant & Covariant indices in Tensors (Symbolic)

[1]:
```
from einsteinpy.symbolic import ChristoffelSymbols, RiemannCurvatureTensor
from einsteinpy.symbolic.predefined import Schwarzschild
import sympy
sympy.init_printing()
```

Analysing the schwarzschild metric along with performing various operations

[2]:
```
sch = Schwarzschild()
sch.tensor()
```

[2]:
$$\begin{bmatrix} 1 - \frac{r_s}{r} & 0 & 0 & 0 \\ 0 & -\frac{1}{c^2\left(1-\frac{r_s}{r}\right)} & 0 & 0 \\ 0 & 0 & -\frac{r^2}{c^2} & 0 \\ 0 & 0 & 0 & -\frac{r^2 \sin^2(\theta)}{c^2} \end{bmatrix}$$

[3]:
```
sch_inv = sch.inv()
sch_inv.tensor()
```

[3]:
$$\begin{bmatrix} \frac{r}{r-r_s} & 0 & 0 & 0 \\ 0 & \frac{c^2(-r+r_s)}{r} & 0 & 0 \\ 0 & 0 & -\frac{c^2}{r^2} & 0 \\ 0 & 0 & 0 & -\frac{c^2}{r^2 \sin^2(\theta)} \end{bmatrix}$$

[4]:
```
sch.order
```

[4]:
2

[5]:
```
sch.config
```

[5]:
```
'll'
```

Obtaining Christoffel Symbols from Metric Tensor

[6]:
```
chr = ChristoffelSymbols.from_metric(sch_inv) # can be initialized from sch also
chr.tensor()
```

[6]:
$$\begin{bmatrix} \begin{bmatrix} 0 & \frac{r_s}{2r^2\left(1-\frac{r_s}{r}\right)} & 0 & 0 \\ \frac{r_s}{2r^2\left(1-\frac{r_s}{r}\right)} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} \frac{r_s c^2\left(1-\frac{r_s}{r}\right)}{2r^2} & 0 & 0 & 0 \\ 0 & -\frac{r_s}{2r^2\left(1-\frac{r_s}{r}\right)} & 0 & 0 \\ 0 & 0 & -r\left(1-\frac{r_s}{r}\right) & 0 \\ 0 & 0 & 0 & -r\left(1-\frac{r_s}{r}\right)\sin^2(\theta) \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{r} & 0 \\ 0 & \frac{1}{r} & 0 & 0 \\ 0 & 0 & 0 & -\sin(\theta)\cos \end{bmatrix} \end{bmatrix}$$

[7]:
```
chr.config
```

[7]:
```
'ull'
```

Changing the first index to covariant

[8]:
```
new_chr = chr.change_config('lll') # changing the configuration to (covariant, covariant, covariant)
new_chr.tensor()
```

[8]:
$$\begin{bmatrix}\begin{bmatrix}0 & \frac{r_s}{2r^2} & 0 & 0\\ \frac{r_s}{2r^2} & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{bmatrix} & \begin{bmatrix}-\frac{r_s}{2r^2} & 0 & 0 & 0\\ 0 & \frac{r_s}{2r^2c^2\left(1-\frac{r_s}{r}\right)^2} & 0 & 0\\ 0 & 0 & \frac{r}{c^2} & 0\\ 0 & 0 & 0 & \frac{r\sin^2(\theta)}{c^2}\end{bmatrix} & \begin{bmatrix}0 & 0 & 0 & 0\\ 0 & 0 & -\frac{r}{c^2} & 0\\ 0 & -\frac{r}{c^2} & 0 & 0\\ 0 & 0 & 0 & \frac{r^2\sin(\theta)\cos(\theta)}{c^2}\end{bmatrix} & \begin{bmatrix}0 & 0 & 0 & \\ 0 & 0 & 0 & \\ 0 & 0 & 0 & \\ 0 & -\frac{r\sin^2(\theta)}{c^2} & -\frac{r^2\sin(\theta)\cos}{c^2} & \end{bmatrix}\end{bmatrix}$$

[9]:
```
new_chr.config
```

[9]:
```
'lll'
```

Any arbitary index configuration would also work!

[10]:
```
new_chr2 = new_chr.change_config('lul')
new_chr2.tensor()
```

[10]:
$$\begin{bmatrix}\begin{bmatrix}0 & \frac{r_s}{2r(r-r_s)} & 0 & 0\\ \frac{r_sc^2(-r+r_s)}{2r^3} & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{bmatrix} & \begin{bmatrix}-\frac{r_s}{2r(r-r_s)} & 0 & 0 & 0\\ 0 & \frac{r_s(-r+r_s)}{2r^3\left(1-\frac{r_s}{r}\right)^2} & 0 & 0\\ 0 & 0 & -\frac{1}{r} & 0\\ 0 & 0 & 0 & -\frac{1}{r}\end{bmatrix} & \begin{bmatrix}0 & 0 & 0 & 0\\ 0 & 0 & r-r_s & 0\\ 0 & \frac{1}{r} & 0 & 0\\ 0 & 0 & 0 & -\frac{\cos(\theta)}{\sin(\theta)}\end{bmatrix} & \begin{bmatrix}0 & 0 & 0 & \\ 0 & 0 & 0 & -(-r\\ 0 & 0 & 0 & \sin\\ 0 & \frac{1}{r} & \frac{\cos(\theta)}{\sin(\theta)} & \end{bmatrix}\end{bmatrix}$$

Obtaining Riemann Tensor from Christoffel Symbols and manipulating it's indices

[11]:
```
rm = RiemannCurvatureTensor.from_christoffels(new_chr2)
rm[0,0,:,:]
```

[11]:
$$\begin{bmatrix}0 & 0 & 0 & 0\\ 0 & -\frac{r_s}{r^2(r-r_s)} & 0 & 0\\ 0 & 0 & \frac{r_s}{2r} & 0\\ 0 & 0 & 0 & \frac{r_s\sin^2(\theta)}{2r}\end{bmatrix}$$

[12]:
```
rm.config
```

[12]:
```
'ulll'
```

[13]:
```
rm2 = rm.change_config("uuuu")
rm2[0,0,:,:]
```

[13]:
$$\begin{bmatrix}0 & 0 & 0 & 0\\ 0 & -\frac{r_sc^4(-r+r_s)^2}{r^3(r-r_s)^2} & 0 & 0\\ 0 & 0 & \frac{r_sc^4}{2r^4(r-r_s)} & 0\\ 0 & 0 & 0 & \frac{r_sc^4}{2r^4(r-r_s)\sin^2(\theta)}\end{bmatrix}$$

[14]:
```
rm3 = rm2.change_config("lulu")
rm3[0,0,:,:]
```

[14]:
$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \frac{r_s c^2 (-r+r_s)^2}{r^3 (r-r_s)^2} & 0 & 0 \\ 0 & 0 & -\frac{r_s c^2 \left(1-\frac{r_s}{r}\right)}{2r^2 (r-r_s)} & 0 \\ 0 & 0 & 0 & -\frac{r_s c^2 \left(1-\frac{r_s}{r}\right)}{2r^2 (r-r_s)} \end{bmatrix}$$

[15]:
```
rm4 = rm3.change_config("ulll")
rm4.simplify()
rm4[0,0,:,:]
```

[15]:
$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -\frac{r_s}{r^2 (r-r_s)} & 0 & 0 \\ 0 & 0 & \frac{r_s}{2r} & 0 \\ 0 & 0 & 0 & \frac{r_s \sin^2(\theta)}{2r} \end{bmatrix}$$

It is seen that `rm` and `rm4` are same as they have the same configuration

1.4.11 Predefined Metrics in Symbolic Module

Importing some of the predefined tensors. All the metrics are comprehensively listed in EinsteinPy documentation.

[1]:
```
from einsteinpy.symbolic.predefined import Schwarzschild, DeSitter, AntiDeSitter,␣
↪Minkowski, find
from einsteinpy.symbolic import RicciTensor, RicciScalar
import sympy
from sympy import simplify

sympy.init_printing()  # for pretty printing
```

Printing the metrics for visualization

All the functions return instances of :py:class:~einsteinpy.symbolic.metric.MetricTensor

[2]:
```
sch = Schwarzschild()
sch.tensor()
```

[2]:
$$\begin{bmatrix} 1-\frac{r_s}{r} & 0 & 0 & 0 \\ 0 & -\frac{1}{c^2 \left(1-\frac{r_s}{r}\right)} & 0 & 0 \\ 0 & 0 & -\frac{r^2}{c^2} & 0 \\ 0 & 0 & 0 & -\frac{r^2 \sin^2(\theta)}{c^2} \end{bmatrix}$$

[3]:
```
Minkowski(c=1).tensor()
```

[3]:
$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 \\ 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 1.0 \end{bmatrix}$$

[4]:
```
DeSitter().tensor()
```

[4]:
$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & e^{\frac{2x}{\alpha}} & 0 & 0 \\ 0 & 0 & e^{\frac{2x}{\alpha}} & 0 \\ 0 & 0 & 0 & e^{\frac{2x}{\alpha}} \end{bmatrix}$$

[5]:
```
AntiDeSitter().tensor()
```

[5]:
$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & \cos^2(t) & 0 & 0 \\ 0 & 0 & \cos^2(t)\sinh^2(\chi) & 0 \\ 0 & 0 & 0 & \sin^2(\theta)\cos^2(t)\sinh^2(\chi) \end{bmatrix}$$

Calculating the scalar (Ricci) curavtures

They should be constant for De-Sitter and Anti-De-Sitter spacetimes.

[6]:
```
scalar_curvature_de_sitter = RicciScalar.from_metric(DeSitter())
scalar_curvature_anti_de_sitter = RicciScalar.from_metric(AntiDeSitter())
```

[7]:
```
scalar_curvature_de_sitter.expr
```

[7]:
$$-\frac{2e^{-\frac{2x}{\alpha}}}{\alpha^2}$$

[8]:
```
scalar_curvature_anti_de_sitter.expr
```

[8]:
$$\frac{\left(\sin^2(t)-1\right)\sinh^2(\chi)-2\cos^2(t)\sinh^2(\chi)}{\cos^2(t)\sinh^2(\chi)}+\frac{\left(\sin^2(t)-1\right)\sin^2(\theta)\sinh^2(\chi)-2\sin^2(\theta)\cos^2(t)\sinh^2(\chi)}{\sin^2(\theta)\cos^2(t)\sinh^2(\chi)}-6$$

On simplifying the expression we got above, we indeed obtain a constant

[9]:
```
simplify(scalar_curvature_anti_de_sitter.expr)
```

[9]:
$$-12$$

Searching for a predefined metric

find function returns a list of available functions

[10]:
```
find("sitter")
```

[10]:
```
['AntiDeSitter', 'AntiDeSitterStatic', 'DeSitter']
```

1.4.12 Ricci Tensor and Scalar Curvature calculations using Symbolic module

[1]:
```
import sympy
from sympy import cos, sin, sinh
from einsteinpy.symbolic import MetricTensor, RicciTensor, RicciScalar
from einsteinpy.symbolic.predefined import AntiDeSitter
sympy.init_printing()
```

Defining the Anti-de Sitter spacetime Metric

[2]:
```
metric = AntiDeSitter()
metric.tensor()
```

[2]:
$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & \cos^2(t) & 0 & 0 \\ 0 & 0 & \cos^2(t)\sinh^2(\chi) & 0 \\ 0 & 0 & 0 & \sin^2(\theta)\cos^2(t)\sinh^2(\chi) \end{bmatrix}$$

Calculating the Ricci Tensor(with both indices covariant)

[3]:
```
Ric = RicciTensor.from_metric(metric)
Ric.tensor()
```

[3]:
$$\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & -3\cos^2(t) & 0 & 0 \\ 0 & 0 & \left(\sin^2(t)-1\right)\sinh^2(\chi) - 2\cos^2(t)\sinh^2(\chi) & 0 \\ 0 & 0 & 0 & \left(\sin^2(t)-1\right)\sin^2(\theta)\sinh^2(\chi) - 2\sin^2(\theta)\cos^2(t)\sinh^2(\chi) \end{bmatrix}$$

Calculating the Ricci Scalar(Scalar Curvature) from the Ricci Tensor

[4]:
```
R = RicciScalar.from_riccitensor(Ric)
R.simplify()
R.expr
```

[4]:
$$-12$$

The curavture is -12 which is in-line with the theoretical results

1.4.13 Symbolically Understanding Christoffel Symbol and Riemann Curvature Tensor using EinsteinPy

[1]:
```
import sympy
from einsteinpy.symbolic import MetricTensor, ChristoffelSymbols,␣
↪RiemannCurvatureTensor

sympy.init_printing()  # enables the best printing available in an environment
```

Defining the metric tensor for 3d spherical coordinates

[2]:
```
syms = sympy.symbols('r theta phi')
# define the metric for 3d spherical coordinates
metric = [[0 for i in range(3)] for i in range(3)]
metric[0][0] = 1
metric[1][1] = syms[0]**2
metric[2][2] = (syms[0]**2)*(sympy.sin(syms[1])**2)
# creating metric object
m_obj = MetricTensor(metric, syms)
m_obj.tensor()
```

[2]:
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & r^2 & 0 \\ 0 & 0 & r^2 \sin^2(\theta) \end{bmatrix}$$

Calculating the christoffel symbols

[3]:
```
ch = ChristoffelSymbols.from_metric(m_obj)
ch.tensor()
```

[3]:
$$\begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & -r & 0 \\ 0 & 0 & -r\sin^2(\theta) \end{bmatrix} & \begin{bmatrix} 0 & \frac{1}{r} & 0 \\ \frac{1}{r} & 0 & 0 \\ 0 & 0 & -\sin(\theta)\cos(\theta) \end{bmatrix} & \begin{bmatrix} 0 & 0 & \frac{1}{r} \\ 0 & 0 & \frac{\cos(\theta)}{\sin(\theta)} \\ \frac{1}{r} & \frac{\cos(\theta)}{\sin(\theta)} & 0 \end{bmatrix} \end{bmatrix}$$

[4]:
```
ch.tensor()[1,1,0]
```

[4]:
$$\frac{1}{r}$$

Calculating the Riemann Curvature tensor

[5]:
```
# Calculating Riemann Tensor from Christoffel Symbols
rm1 = RiemannCurvatureTensor.from_christoffels(ch)
rm1.tensor()
```

[5]:
$$\begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{bmatrix}$$

[6]:
```
# Calculating Riemann Tensor from Metric Tensor
rm2 = RiemannCurvatureTensor.from_metric(m_obj)
rm2.tensor()
```

[6]:

$$\left[\begin{matrix}\left[\begin{matrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{matrix}\right] & \left[\begin{matrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{matrix}\right] & \left[\begin{matrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{matrix}\right]\\\left[\begin{matrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{matrix}\right] & \left[\begin{matrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{matrix}\right] & \left[\begin{matrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{matrix}\right]\\\left[\begin{matrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{matrix}\right] & \left[\begin{matrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{matrix}\right] & \left[\begin{matrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{matrix}\right]\end{matrix}\right]$$

Calculating the christoffel symbols for Schwarzschild Spacetime Metric

- The expressions are unsimplified

[7]:

```
syms = sympy.symbols("t r theta phi")
G, M, c, a = sympy.symbols("G M c a")
# using metric values of schwarzschild space-time
# a is schwarzschild radius
list2d = [[0 for i in range(4)] for i in range(4)]
list2d[0][0] = 1 - (a / syms[1])
list2d[1][1] = -1 / ((1 - (a / syms[1])) * (c ** 2))
list2d[2][2] = -1 * (syms[1] ** 2) / (c ** 2)
list2d[3][3] = -1 * (syms[1] ** 2) * (sympy.sin(syms[2]) ** 2) / (c ** 2)
sch = MetricTensor(list2d, syms)
sch.tensor()
```

[7]:

$$\left[\begin{matrix}- \frac{a}{r} + 1 & 0 & 0 & 0\\0 & - \frac{1}{c^{2} \left(- \frac{a}{r} + 1\right)} & 0 & 0\\0 & 0 & - \frac{r^{2}}{c^{2}} & 0\\0 & 0 & 0 & - \frac{r^{2} \sin^{2}{\left(\theta \right)}}{c^{2}}\end{matrix}\right]$$

[8]:

```
# single substitution
subs1 = sch.subs(a,0)
subs1.tensor()
```

[8]:

$$\left[\begin{matrix}1 & 0 & 0 & 0\\0 & - \frac{1}{c^{2}} & 0 & 0\\0 & 0 & - \frac{r^{2}}{c^{2}} & 0\\0 & 0 & 0 & - \frac{r^{2} \sin^{2}{\left(\theta \right)}}{c^{2}}\end{matrix}\right]$$

[9]:

```
# multiple substitution
subs2 = sch.subs([(a,0), (c,1)])
subs2.tensor()
```

[9]:

$$\left[\begin{matrix}1 & 0 & 0 & 0\\0 & -1 & 0 & 0\\0 & 0 & - r^{2} & 0\\0 & 0 & 0 & - r^{2} \sin^{2}{\left(\theta \right)}\end{matrix}\right]$$

[10]:

```
sch_ch = ChristoffelSymbols.from_metric(sch)
sch_ch.tensor()
```

[10]:
$$\left[\begin{bmatrix} 0 & \frac{a}{2r^2\left(-\frac{a}{r}+1\right)} & 0 & 0 \\ \frac{a}{2r^2\left(-\frac{a}{r}+1\right)} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{ac^2\left(-\frac{a}{r}+1\right)}{2r^2} & 0 & 0 & 0 \\ 0 & -\frac{a}{2r^2\left(-\frac{a}{r}+1\right)} & 0 & 0 \\ 0 & 0 & -r\left(-\frac{a}{r}+1\right) & 0 \\ 0 & 0 & 0 & -r\left(-\frac{a}{r}+1\right)\sin^2(\theta) \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{r} & 0 \\ 0 & \frac{1}{r} & 0 & 0 \\ 0 & 0 & 0 & -\sin(\theta) \end{bmatrix}$$

Calculating the simplified expressions

[11]:
```
simplified = sch_ch.simplify()
simplified
```

[11]:
$$\left[\begin{bmatrix} 0 & \frac{a}{2r(-a+r)} & 0 & 0 \\ \frac{a}{2r(-a+r)} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{ac^2(-a+r)}{2r^3} & 0 & 0 & 0 \\ 0 & \frac{a}{2r(a-r)} & 0 & 0 \\ 0 & 0 & a-r & 0 \\ 0 & 0 & 0 & (a-r)\sin^2(\theta) \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{r} & 0 \\ 0 & \frac{1}{r} & 0 & 0 \\ 0 & 0 & 0 & -\frac{\sin(2\theta)}{2} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & \\ 0 & 0 & 0 & \\ 0 & 0 & 0 & \\ 0 & \frac{1}{r} & \frac{1}{\tan(\theta)} & \end{bmatrix}$$

1.4.14 Weyl Tensor calculations using Symbolic module

[1]:
```
import sympy
from sympy import cos, sin, sinh
from einsteinpy.symbolic import MetricTensor, WeylTensor

sympy.init_printing()
```

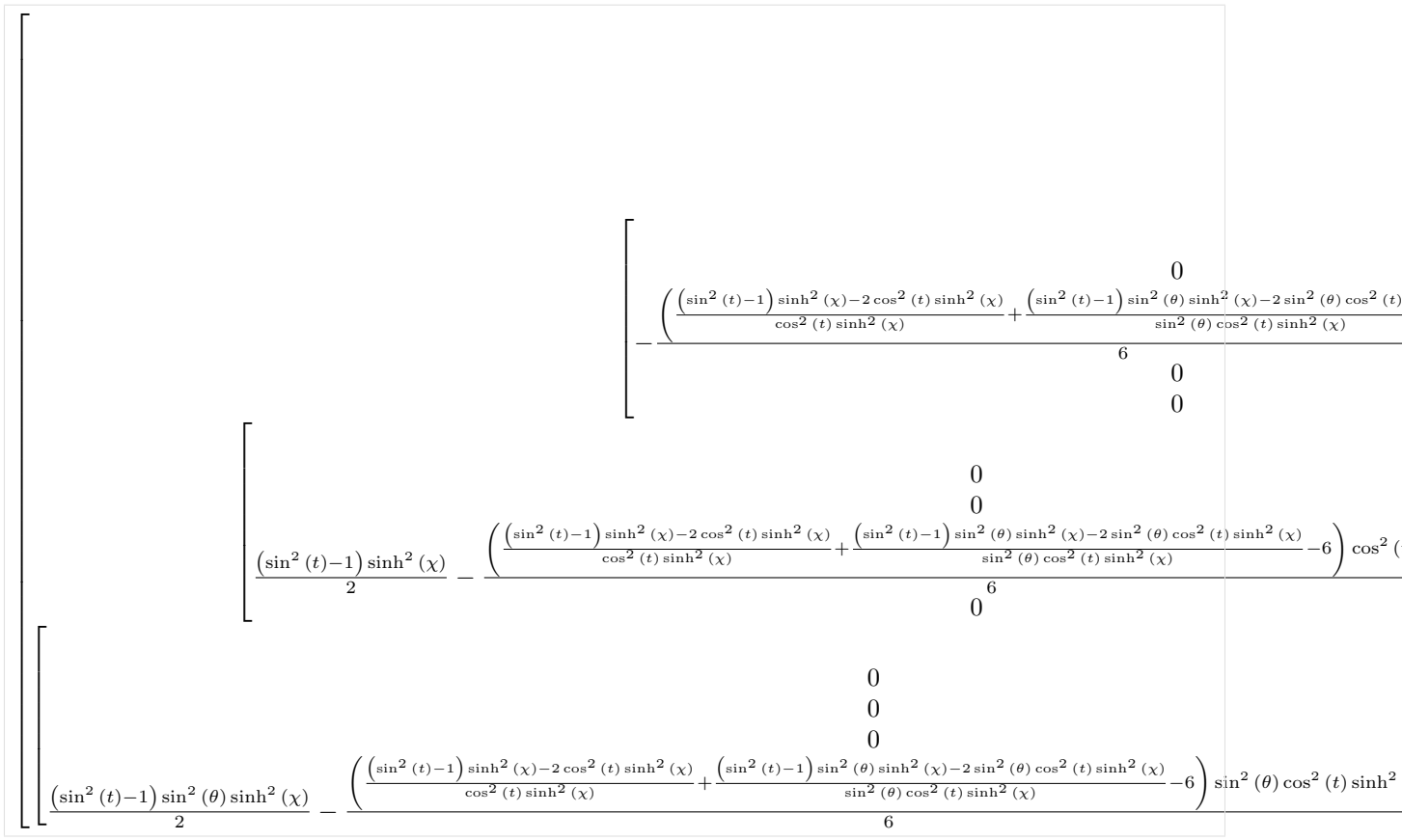
Defining the Anti-de Sitter spacetime Metric

[2]:
```
syms = sympy.symbols("t chi theta phi")
t, ch, th, ph = syms
m = sympy.diag(-1, cos(t) ** 2, cos(t) ** 2 * sinh(ch) ** 2, cos(t) ** 2 * sinh(ch)
↪** 2 * sin(th) ** 2).tolist()
metric = MetricTensor(m, syms)
```

Calculating the Weyl Tensor (with all indices covariant)

[3]:
```
weyl = WeylTensor.from_metric(metric)
weyl.tensor()
```

[3]:



[4]:
```
weyl.config
```

[4]:
```
'llll'
```

1.5 What's new

1.5.1 einsteinpy 0.3.1 - 2021-01-16

This release is a minor patch release for fixing a minor Debian issue.

Contributors

This is the complete list of the people that contributed to this release, with a + sign indicating first contribution.

- Shreyas Bapat

1.5.2 einsteinpy 0.3.0 - 2020-05-05

This major release would bring some very important improvements. This release fixes a very crucial bug with sympy. Fixes coordinate conversions so they don't fail on edge cases anymore.

EinsteinPy now uses GitHub Actions for macOS builds. Big changes to the plotting module.

The release comes for the paper of EinsteinPy. The release marks the beginning of Google Summer of Code 2020. The release also brings a new rays module, which will form the base for null geodesics in future releases.

Features

- Loads of Predefined Metrics
- Sympy version incompatibilities handled
- Numba as a default installation
- Lorentz Transform for Einstein Tensor
- Lorentz Transform to Tensor Class
- Hypersurface Plotting API similar to the common plotting API
- Find Function in Predefined Metrics
- Increased Code Coverage
- New rays module
- Plotting Black Hole Shadows
- Coordinate Subscripting
- Supports Python 3.8, dropping support fpr Python 3.5
- numpy moveaxis replaced with sympy permutedims
- name parameter in Metric Tensor
- Tags to Tensor names

Contributors

This is the complete list of the people that contributed to this release, with a + sign indicating first contribution.

- Shreyas Bapat
- Ritwik Saha
- Manvi Gupta
- Micky Yun Chan+
- DylanBrdt+ (GitHub Username)
- Vineet Gandham+
- Pratyush Kerhalkar+
- Bhavam Vidyarthi+
- Khalid Shaikh+
- Rohit Sanjay+

- Saurabh+
- Raahul Singh+
- Nimesh Vashishtha+
- Shamanth R Nayak K+
- Arnav Das+
- Gim Seng Ng+
- Nihar Gupte+
- Suyash Salampuria+
- Atul Mangat+
- Ganesh Tarone+
- Shreyas Kalvankar+
- Swastik Singh+
- Jyotirmaya Shivottam+
- Sitara Srinivasan+
- Aayush Gautam+
- Zac Yauney+
- Gagan-Shenoy+
- Bibek Gautam+
- Erin Allard+
- Suyog Garg+

1.5.3 einsteinpy 0.2.1 - 2019-11-02

This minor release would bring improvements and new feature additions to the already existing symbolic calculations module along with performance boosts of order of 15x.

This release concludes the SOCIS 2019 projects of Sofía Ortín Vela (ortinvela.sofia@gmail.com) and Varun Singh(varunsinghs2021@gmail.com).

Part of this release is sponsored by European Space Agency, through Summer of Code in Space (SOCIS) 2019 program.

Features

- New tensors in symbolic module
  - Ricci Scalar
  - Weyl Tensor
  - Stress-Energy-Momentum Tensor
  - Einstein Tensor
  - Schouten Tensor
- Improvement in performance of current tensors

- Lambdify option for tensors
- Support for vectors at arbitrary space-time symbolically as 1^{st} order tensor.
- Support for scalars at arbitrary space-time symbolically as 0^{th} order tensor.
- Addition of constants sub-module to symbolic module
- Improvement in speed of Geodesic plotting
- Move away from Jupyter and Plotly Widgets
- New Plotting Framework

Contributors

This is the complete list of the people that contributed to this release, with a + sign indicating first contribution.

- Shreyas Bapat
- Ritwik Saha
- Sofía Ortín Vela
- Varun Singh
- Arnav Das+
- Calvin Jay Ross+

1.5.4 einsteinpy 0.2.0 - 2019-07-15

This release brings a lot of new features for the EinsteinPy Users.

A better API, intuitive structure and easy coordinates handling! This major release comes before Python in Astronomy 2019 workshop and brings a lots of cool stuff.

Part of this release is sponsored by ESA/ESTEC – Adv. Concepts & Studies Office (European Space Agency), through Summer of Code in Space (SOCIS) 2019 program.

This is a short-term supported version and will be supported only until December 2019. For any feature request, write a mail to developers@einsteinpy.org describing what you need.

Features

- Kerr Metric
- Kerr-Newman Metric
- Coordinates Module with Boyer Lindquist Coordinates and transformation
- Bodies Module
- Defining Geodesics with ease!
- Animated plots
- Intuitive API for plotting
- Schwarzschild Hypersurface Embedding
- Interactive Plotting
- Environment-aware plotting and exceptional support for iPython Notebooks!

- Support for Tensor Algebra in General Relativity
- Symbolic Manipulation of Metric Tensor, Riemann Tensor and Ricci Tensor
- Support for Index Raising and Lowering in Tensors
- Numerical Calculation and Symbolic Manipulation of Christoffel Symbols
- Calculations of Event Horizon and Ergosphere of Kerr Black holes!

Contributors

This is the complete list of the people that contributed to this release, with a + sign indicating first contribution.

- Shreyas Bapat
- Ritwik Saha
- Bhavya Bhatt
- Sofía Ortín Vela+
- Raphael Reyna+
- Prithvi Manoj Krishna+
- Manvi Gupta+
- Divya Gupta+
- Yash Sharma+
- Shilpi Jain+
- Rishi Sharma+
- Varun Singh+
- Alpesh Jamgade+
- Saurabh Bansal+
- Tanmay Rustagi+
- Abhijeet Manhas+
- Ankit Khandelwal+
- Tushar Tyagi+
- Hrishikesh Sarode
- Naman Tayal+
- Ratin Kumar+
- Govind Dixit+
- Jialin Ma+

Bugs Fixed

- Issue #115: Coordinate Conversion had naming issues that made them confusing!
- Issue #185: Isort had conflicts with Black
- Issue #210: Same notebook had two different listings in Example Gallery
- Issue #264: Removing all relative imports
- Issue #265: New modules were lacking API Docs
- Issue #266: The logo on documentation was not rendering
- Issue #267: Docs were not present for Ricci Tensor and Vacuum Metrics
- Issue #277: Coordinate Conversion in plotting module was handled incorrectly

Backwards incompatible changes

- The old `StaticGeodesicPlotter` has been renamed to `einsteinpy.plotting.senile.StaticGeodesicPlotter`, please adjust your imports accordingly
- The old `ScatterGeodesicPlotter` has been renamed to `einsteinpy.plotting.senile.ScatterGeodesicPlotter`, please adjust your imports accordingly.
- `einsteinpy.metric.Schwarzschild`, `einsteinpy.metric.Kerr`, and `einsteinpy.metric.KerrNewman` now have different signatures for class methods, and they now explicitly support `einsteinpy.coordinates` coordinate objects. Check out the notebooks and their respective documentation.
- The old *coordinates* conversion in `einsteinpy.utils` has been deprecated.
- The old *symbolic* module in `einsteinpy.utils` has been moved to `einsteinpy.symbolic`.

1.5.5 einsteinpy 0.1.0 - 2019-03-08

This is a major first release for world's first actively maintained python library for General Relativity and Numerical methods. This major release just comes before the Annual AstroMeet of IIT Mandi, AstraX. This will be a short term support version and will be supported only until late 2019.

Features

- Schwarzschild Geometry Analysis and trajectory calculation
- Symbolic Calculation of various tensors in GR
- Christoffel Symbols
- Riemann Curvature Tensor
- Static Geodesic Plotting
- Velocity of Coordinate time w.r.t proper time
- Easy Calculation of Schwarzschild Radius
- Coordinate conversion with unit handling
- Spherical/Cartesian Coordinates
- Boyer-Lindquist/Cartesian Coordinates

Contributors

This is the complete list of the people that contributed to this release, with a + sign indicating first contribution.

- Shreyas Bapat+
- Ritwik Saha+
- Bhavya Bhatt+
- Priyanshu Khandelwal+
- Gaurav Kumar+
- Hrishikesh Sarode+
- Sashank Mishra+

1.6 Developer Guide

Einsteinpy is a community project, hence all contributions are more than welcome!

1.6.1 Bug reporting

Not only things break all the time, but also different people have different use cases for the project. If you find anything that doesn't work as expected or have suggestions, please refer to the issue tracker on GitHub.

1.6.2 Documentation

Documentation can always be improved and made easier to understand for newcomers. The docs are stored in text files under the *docs/source* directory, so if you think anything can be improved there please edit the files and proceed in the same way as with *code writing*.

The Python classes and methods also feature inline docs: if you detect any inconsistency or opportunity for improvement, you can edit those too.

Besides, the wiki is open for everybody to edit, so feel free to add new content.

To build the docs, you must first create a development environment (see below) and then in the `docs/` directory run:

```
$ cd docs
$ make html
```

After this, the new docs will be inside `build/html`. You can open them by running an HTTP server:

```
$ cd build/html
$ python -m http.server
Serving HTTP on 0.0.0.0 port 8000 ...
```

And point your browser to http://0.0.0.0:8000.

1.6.3 Code writing

Code contributions are welcome! If you are looking for a place to start, help us fixing bugs in einsteinpy and check out the "easy" tag. Those should be easier to fix than the others and require less knowledge about the library.

If you are hesitant on what IDE or editor to use, just choose one that you find comfortable and stick to it while you are learning. People have strong opinions on which editor is better so I recommend you to ignore the crowd for the time being - again, choose one that you like :)

If you ask me for a recommendation, I would suggest PyCharm (complete IDE, free and gratis, RAM-hungry) or vim (powerful editor, very lightweight, steep learning curve). Other people use Spyder, emacs, gedit, Notepad++, Sublime, Atom...

You will also need to understand how git works. git is a decentralized version control system that preserves the history of the software, helps tracking changes and allows for multiple versions of the code to exist at the same time. If you are new to git and version control, I recommend following the Try Git tutorial.

If you already know how all this works and would like to contribute new features then that's awesome! Before rushing out though please make sure it is within the scope of the library so you don't waste your time - email us or chat with us on Riot!.

If the feature you suggest happens to be useful and within scope, you will probably be advised to create a new wiki page with some information about what problem you are trying to solve, how do you plan to do it and a sketch or idea of how the API is going to look like. You can go there to read other good examples on how to do it. The purpose is to describe without too much code what you are trying to accomplish to justify the effort and to make it understandable to a broader audience.

All new features should be thoroughly tested, and in the ideal case the coverage rate should increase or stay the same. Automatic services will ensure your code works on all the operative systems and package combinations einsteinpy support - specifically, note that einsteinpy is a Python 3 only library.

1.6.4 Development environment

These are some succint steps to set up a development environment:

1. Install git on your computer.
2. Register to GitHub.
3. Fork einsteinpy.
4. Clone your fork.
5. Install it in development mode using `pip install --editable /path/to/einsteinpy/[dev]` (this means that the installed code will change as soon as you change it in the download location).
6. Create a new branch.
7. Make changes and commit.
8. Push to your fork.
9. Open a pull request!

1.6.5 Code Linting

To get the quality checks passed, the code should follow some standards listed below:

1. Black for code formatting.
2. isort for imports sorting.
3. mypy for static type checking.

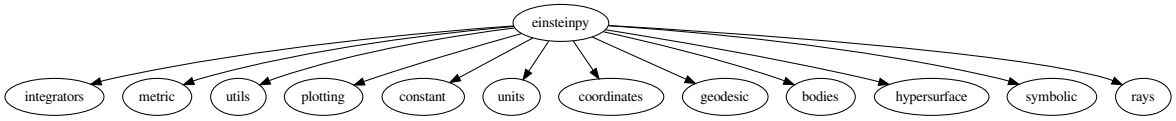
But to avoid confusion, we have setup tox for doing this in one command and doing it properly! Run:

```
$ cd einsteinpy/
$ tox -e reformat
```

And it will format all your code!

1.7 EinsteinPy API

Welcome to the API documenatation of EinsteinPy. Please navigate through the given modules to get to know the API of the classes and methods. If you find anything missing, please open an issue in the repo .



1.7.1 Integrators module

This module contains the methods of solving Ordinary Differential Equations.

Runge Kutta module

class einsteinpy.integrators.runge_kutta.**RK4naive**(*fun*, *t0*, *y0*, *t_bound*, *stepsize*)

Bases: object

Class for Defining Runge-Kutta 4th Order ODE solving method

Initialization

Parameters

- **fun** (*function*) – Should accept t, y as parameters, and return same type as y
- **t0** (*float*) – Initial t
- **y0** (*array or float*) – Initial y
- **t_bound** (*float*) – Boundary time - the integration won't continue beyond it. It also determines the direction of the integration.
- **stepsize** (*float*) – Size of each increment in t

step()

Updates the value of self.t and self.y

class einsteinpy.integrators.runge_kutta.**RK45**(*fun*, *t0*, *y0*, *t_bound*, *stepsize*, *rtol=None*, *atol=None*)

Bases: scipy.integrate._ivp.rk.RK45

This Class inherits ~scipy.integrate.RK45 Class

Initialization

Parameters

- **fun** (*function*) – Should accept t, y as parameters, and return same type as y
- **t0** (*float*) – Initial t
- **y0** (*array or float*) – Initial y
- **t_bound** (*float*) – Boundary time - the integration won't continue beyond it. It also determines the direction of the integration.
- **stepsize** (*float*) – Size of each increment in t
- **rtol** (*float*) – Relative tolerance, defaults to 0.2*stepsize
- **atol** (*float*) – Absolute tolerance, defaults to rtol/0.8e3

step()

Updates the value of self.t and self.y

1.7.2 Metric module

This module contains the basic classes of different metrics for various space-time geometries including schwarzschild, kerr etc.

base-metric module

This module contains the class, that defines a general spacetime. All other metric classes derive from it. It has methods, that can be used as utility functions. Users are recommended to inherit this class to create user-defined metric classes.

Docstring for base_metric.py module

This module defines the BaseMetric class, which is the base class for all metrics in EinsteinPy. This class contains several utilities, that are used in einsteinpy.metric to define classes for vacuum solutions. Users are advised to inherit this class, while defining their own metric classes. Two parameters to note are briefly described below:

- metric_cov: User-supplied function, defining the covariant form of the metric tensor. Users need to supply just this to completely determine the metric tensor, as the contravariant form is calculated and accessed through a predefined method, metric_contravariant().
- perturbation: User-supplied function, defining a perturbation to the metric. Currently, no checks are performed to ascertain the physicality of the resulting perturbed metric. Read the documentation on metric_covariant() below, to learn more.

Also, note that, users should call metric_covariant to access the perturbed, covariant form of the metric. For unperturbed underlying metric, users should call metric_cov, which returns the metric, that they had supplied.

class einsteinpy.metric.base_metric.**BaseMetric**(*coords*, *M*, *a=<Quantity 0.>*, *Q=<Quantity 0. C>*, *name='Base Metric'*, *metric_cov=None*, *christoffels=None*, *f_vec=None*, *perturbation=None*)

Bases: object

For defining a general Metric

Constructor

Parameters

- **coords** (***) – Coordinate system, in which Metric is to be represented
- **M** (*Quantity*) – Mass of gravitating body, e.g. Black Hole
- **a** (*Quantity*) – Dimensionless Spin Parameter Defaults to `0`
- **Q** (*Quantity*) – Charge on gravitating body, e.g. Black Hole Defaults to `0 C`
- **name** (*str, optional*) – Name of the Metric Tensor. Defaults to `"Base Metric"`
- **metric_cov** (*callable, optional*) – Function, defining Covariant Metric Tensor It should return a real-valued tensor (2D Array), at supplied coordinates Defaults to `None` Consult pre-defined classes for function definition
- **christoffels** (*callable, optional*) – Function, defining Christoffel Symbols It should return a real-valued (4,4,4) array, at supplied coordinates Defaults to `None` Consult pre-defined classes for function definition
- **f_vec** (*callable, optional*) – Function, defining RHS of Geodesic Equation It should return a real-valued (8) vector, at supplied coordinates Defaults to `None` Consult pre-defined classes for function definition
- **perturbation** (*callable, optional*) – Function, defining Covariant Perturbation Tensor It should return a real-valued tensor (2D Array), at supplied coordinates Defaults to `None` Function definition similar to `metric_cov`

static sigma(*r*, *theta*, *M*, *a*)

Returns the value of `r**2 + alpha**2 * cos(theta)**2` Specific to Boyer-Lindquist coordinates Applies to Kerr Geometry

Parameters

- **r** (*float or Quantity*) – r-component of 4-Position
- **theta** (*float or Quantity*) – theta-component of 4-Position
- **M** (*float or Quantity*) – Mass of gravitating body
- **a** (*float or Quantity*) – Spin Parameter

Returns The value of `r**2 + alpha**2 * cos(theta)**2`

Return type float

static delta(*r*, *M*, *a*, *Q=0*)

Returns the value of `r**2 - r_s * r + alpha**2 + r_Q**2` Specific to Boyer-Lindquist coordinates Applies to Kerr & Kerr-Newman Geometries

Parameters

- **r** (*float or Quantity*) – r-component of 4-Position
- **M** (*float or Quantity*) – Mass of gravitating body
- **a** (*float or Quantity*) – Spin Parameter
- **Q** (*float or Quantity, optional*) – Charge on gravitating body Defaults to `0` (for Kerr Geometry)

Returns The value of `r**2 - r_s * r + alpha**2 + r_Q**2`

Return type float

static rho(*r*, *theta*, *M*, *a*)

Returns the value of `sqrt(r**2 + alpha**2 * cos(theta)**2) == sqrt(sigma)` Specific to Boyer-Lindquist coordinates Applies to Kerr-Newman Geometry

Parameters

- **r** (*float or Quantity*) – r-component of 4-Position
- **theta** (*float or Quantity*) – theta-component of 4-Position
- **M** (*float or Quantity*) – Mass of gravitating body
- **a** (*float or Quantity*) – Spin Parameter

Returns The value of `sqrt(r**2 + alpha**2 * cos(theta)**2) == sqrt(sigma)`

Return type float

static schwarzschild_radius(*M*)

Returns Schwarzschild Radius

Parameters **M** (*float or Quantity*) – Mass of gravitating body

Returns Schwarzschild Radius for a given mass

Return type float

static alpha(*M*, *a*)

Returns Rotational Length Parameter (alpha) that is used in the Metric. Following equations are relevant: alpha = J / Mc a = Jc / GM**2 alpha = GMa / c**2 where, 'a' is the dimensionless Spin Parameter (0 <= a <= 1)

Parameters

- **M** (*float or Quantity*) – Mass of gravitating body
- **a** (*float or Quantity*) – Number between 0 and 1

Returns Rotational Length Parameter

Return type float

Raises **ValueError** – If a is not between 0 and 1

singularities()

Returns the Singularities of the Metric Depends on the choice of Coordinate Systems Applies to Kerr and Kerr-Newman Geometries

Returns Dictionary of singularities in the geometry `{ "inner_ergosphere": function(theta), "inner_horizon": float, "outer_horizon": float, "outer_ergosphere": function(theta) }`

Return type dict

Raises ***CoordinateError*** – If `einsteinpy.metric.*` does not have the metric in the coordinate system, the metric object has been instantiated with

metric_covariant(*x_vec*)

Returns Covariant Metric Tensor Adds Kerr-Schild (Linear) Perturbation to metric, if `perturbation` is defined in Metric object Currently, this does not consider Gauge Fixing or any physical checks for the `perturbation` matrix. Please exercise caution while using `perturbation`.

Parameters **x_vec** (*array_like*) – Position 4-Vector

Returns Covariant Metric Tensor Numpy array of shape (4,4)

Return type ndarray

metric_contravariant(*x_vec*)
Returns Contravariant Metric Tensor Adds Kerr-Schild (Linear) Perturbation to metric, if `perturbation` is not None in Metric object Currently, this does not consider Gauge Fixing or any physical checks for the `perturbation` matrix. Please exercise caution while using `perturbation`.

Parameters **x_vec** (*array_like*) – Position 4-Vector

Returns Contravariant Metric Tensor

Return type ndarray

calculate_trajectory(*start_lambda=0.0*, *end_lambda=10.0*, *stop_on_singularity=True*, *OdeMethodKwargs={'stepsize': 0.001}*, *return_cartesian=False*)
Deprecated in 0.4.0. Please use `einsteinpy.Geodesic`.

Calculate trajectory in manifold according to geodesic equation

schwarzschild module

This module contains the class, defining Schwarzschild Spacetime:

class einsteinpy.metric.schwarzschild.**Schwarzschild**(*coords*, *M*)
Bases: *einsteinpy.metric.base_metric.BaseMetric*

Class for defining Schwarzschild Geometry

Constructor

Parameters

- **coords** (*) – Coordinate system, in which Metric is to be represented
- **M** (*Quantity*) – Mass of gravitating body, e.g. Black Hole

metric_covariant(*x_vec*)
Returns Covariant Schwarzschild Metric Tensor in chosen Coordinates

Parameters **x_vec** (*array_like*) – Position 4-Vector

Returns Covariant Schwarzschild Metric Tensor in chosen Coordinates Numpy array of shape (4,4)

Return type ndarray

kerr module

This module contains the class, defining Kerr Spacetime:

class einsteinpy.metric.kerr.**Kerr**(*coords*, *M*, *a*)
Bases: *einsteinpy.metric.base_metric.BaseMetric*

Class for defining the Kerr Geometry

Constructor

Parameters

- **coords** (*) – Coordinate system, in which Metric is to be represented
- **M** (*Quantity*) – Mass of gravitating body, e.g. Black Hole

- **a** (*Quantity*) – Spin Parameter

metric_covariant(*x_vec*)
Returns Covariant Kerr Metric Tensor in chosen Coordinates

Parameters **x_vec** (*array_like*) – Position 4-Vector

Returns Covariant Kerr Metric Tensor in chosen Coordinates Numpy array of shape (4,4)

Return type ndarray

Raises ***CoordinateError*** – Raised, if the metric is not available in the supplied Coordinate System

static nonzero_christoffels()
Returns a list of tuples consisting of indices of non-zero Christoffel Symbols in Kerr Metric, computed in real-time

Returns List of tuples Each tuple (i,j,k) represents Christoffel Symbols, with i as upper index and j, k as lower indices.

Return type list

kerr-newman module

This module contains the class, defining Kerr-Newman Spacetime:

class einsteinpy.metric.kerrnewman.**KerrNewman**(*coords*, *M*, *a*, *Q*, *q=<Quantity 0. C / kg>*)
Bases: *einsteinpy.metric.base_metric.BaseMetric*

Class for defining Kerr-Newman Goemetry

Constructor

Parameters

- **coords** (*) – Coordinate system, in which Metric is to be represented
- **M** (*Quantity*) – Mass of gravitating body, e.g. Black Hole
- **a** (*Quantity*) – Spin Parameter
- **Q** (*Quantity*) – Charge on gravitating body, e.g. Black Hole
- **q** (*Quantity, optional*) – Charge, per unit mass, of the test particle Defaults to `0 C / kg`

metric_covariant(*x_vec*)
Returns Covariant Kerr-Newman Metric Tensor in chosen Coordinates

Parameters **x_vec** (*array_like*) – Position 4-Vector

Returns Covariant Kerr-Newman Metric Tensor in chosen Coordinates Numpy array of shape (4,4)

Return type ndarray

Raises ***CoordinateError*** – Raised, if the metric is not available in the supplied Coordinate System

em_potential_covariant(*x_vec*)
Returns Covariant Electromagnetic 4-Potential Specific to Kerr-Newman Geometries

Parameters **x_vec** (*array_like*) – Position 4-Vector

Returns Covariant Electromagnetic 4-Potential Numpy array of shape (4,)

Return type ndarray

em_potential_contravariant(*x_vec*)
Returns Contravariant Electromagnetic 4-Potential Specific to Kerr-Newman Geometries

Parameters **x_vec** (*array_like*) – Position 4-Vector

Returns Contravariant Electromagnetic 4-Potential Numpy array of shape (4,)

Return type ndarray

em_tensor_covariant(*x_vec*)
Returns Covariant Electromagnetic Tensor Specific to Kerr-Newman Geometries

Parameters **x_vec** (*array_like*) – Position 4-Vector

Returns Covariant Electromagnetic Tensor Numpy array of shape (4, 4)

Return type ndarray

em_tensor_contravariant(*x_vec*)
Returns Contravariant Electromagnetic Tensor Specific to Kerr-Newman Geometries

Parameters **x_vec** (*array_like*) – Position 4-Vector

Returns Contravariant Electromagnetic Tensor Numpy array of shape (4, 4)

Return type ndarray

1.7.3 Symbolic Relativity Module

This module contains the classes for performing symbolic calculations related to General Relativity.

Predefined Metrics

This module contains various pre-defined space-time metrics in General Relativity.

All Available Metrics

All the currently available pre-defined metrics are listed here.

Minkowski Space-Time

einsteinpy.symbolic.predefined.minkowski.**MinkowskiCartesian**(*c=c*)
Minkowski(flat) space-time in Cartesian coordinates. Space-time without any curvature or matter.

Parameters **c** (*Basic or int or float*) – Any value to assign to speed of light. Defaults to 'c'.

einsteinpy.symbolic.predefined.minkowski.**Minkowski**(*c=c*)
Minkowski(flat) space-time in Cartesian coordinates. Space-time without any curvature or matter.

Parameters **c** (*Basic or int or float*) – Any value to assign to speed of light. Defaults to 'c'.

einsteinpy.symbolic.predefined.minkowski.**MinkowskiPolar**(*c=c*)
Minkowski(flat) space-time in Polar coordinates. Space-time without any curvature or matter.

Parameters **c** (*Basic or int or float*) – Any value to assign to speed of light. Defaults to 'c'.

Vacuum Solutions

einsteinpy.symbolic.predefined.vacuum_solutions.**Schwarzschild**(*c=c*, *sch=r_s*)

Schwarzschild exterior metric in curvature coordinates Schwarzschild, Sitz. Preuss. Akad. Wiss., p189, (1916) Stephani (13.19) p157

Parameters

- **c** (*Basic or int or float*) – Any value to assign to speed of light. Defaults to c.
- **sch** (*Basic or int or float*) – Any value to assign to Schwarzschild Radius of the central object. Defaults to r_s.

einsteinpy.symbolic.predefined.vacuum_solutions.**Kerr**(*c=c*, *sch=r_s*, *a=a*)

Kerr Metric in Boyer Lindquist coordinates.

Parameters

- **c** (*Basic or int or float*) – Any value to assign to speed of light. Defaults to c.
- **sch** (*Basic or int or float*) – Any value to assign to Schwarzschild Radius of the central object. Defaults to r_s.
- **a** (*Basic or int or float*) – Spin factor of the heavy body. Usually, given by J/(Mc), where J is the angular momentum. Defaults to a.

einsteinpy.symbolic.predefined.vacuum_solutions.**KerrNewman**(*c=c*, *G=G*, *eps_0=eps_0*, *sch=r_s*, *a=a*, *Q=Q*)

Kerr-Newman Metric in Boyer Lindquist coordinates.

Parameters

- **c** (*Basic or int or float*) – Any value to assign to speed of light. Defaults to c.
- **G** (*Basic or int or float*) – Any value to assign to the Newton's (or gravitational) constant. Defaults to G.
- **eps_0** (*Basic or int or float*) – Any value to assign to the electric constant or permittivity of free space. Defaults to eps_0.
- **sch** (*Basic or int or float*) – Any value to assign to Schwarzschild Radius of the central object. Defaults to r_s.
- **a** (*Basic or int or float*) – Spin factor of the heavy body. Usually, given by J/(Mc), where J is the angular momentum. Defaults to a.
- **Q** (*Basic or int or float*) – Any value to assign to eletric charge of the central object. Defaults to Q.

einsteinpy.symbolic.predefined.vacuum_solutions.**ReissnerNordstorm**(*c=c*, *G=G*, *eps_0=eps_0*, *sch=r_s*, *Q=Q*)

The Reissner–Nordström metric in spherical coordinates A static solution to the Einstein–Maxwell field equations, which corresponds to the gravitational field of a charged, non-rotating, spherically symmetric body of mass M.

Parameters

- **c** (*Basic or int or float*) – Any value to assign to speed of light. Defaults to `c`.
- **G** (*Basic or int or float*) – Any value to assign to the Newton's (or gravitational) constant. Defaults to `G`.
- **eps_0** (*Basic or int or float*) – Any value to assign to the electric constant or permittivity of free space. Defaults to `eps_0`.
- **sch** (*Basic or int or float*) – Any value to assign to Schwarzschild Radius of the central object. Defaults to `r_s`.
- **Q** (*Basic or int or float*) – Any value to assign to eletric charge of the central object. Defaults to `Q`.

De Sitter and Anti De Sitter

This module contains pre-defined functions to obtain instances of various forms of Anti-De-Sitter and De-Sitter space-times.

einsteinpy.symbolic.predefined.de_sitter.**AntiDeSitter**()
Anti-de Sitter space

Hawking and Ellis (5.9) p131

einsteinpy.symbolic.predefined.de_sitter.**AntiDeSitterStatic**()
Static form of Anti-de Sitter space

Hawking and Ellis (5.9) p131

einsteinpy.symbolic.predefined.de_sitter.**DeSitter**()
de Sitter space

Hawking and Ellis p125

C-Metric

einsteinpy.symbolic.predefined.cmetric.**CMetric**()
The C-metric Stephani (Table 16.2) p188

Godel

einsteinpy.symbolic.predefined.godel.**Godel**()
Godel metric Rev. Mod. Phys., v21, p447, (1949) Stephani (10.25) 122

Davidson

einsteinpy.symbolic.predefined.davidson.**Davidson**()
Davidson's cylindrically symmetric radiation perfect fluid universe Davidson, J. Math. Phys., v32, p1560, (1991)

Bessel Gravitational Wave

einsteinpy.symbolic.predefined.bessel_gravitational_wave.**BesselGravitationalWave**(*C=C*)

Exact gravitational wave solution without diffraction. Class. Quantum Grav., 16:L75–78, 1999. D. Kramer.

An exact solution describing an axisymmetric gravitational wave propagating in the z-direction in closed form. This solution to Einstein's vacuum field equations has the remarkable property that the curvature invariants decrease monotonically with increasing radial distance from the axis and vanish at infinity. The solution is regular at the symmetry axis.

Parameters **C** (*Basic or int or float*) – Constant for Bessel metric, the choice of the constant is not really relavent for details see the paper. Defaults to 'C'.

Barriola Vilekin

einsteinpy.symbolic.predefined.barriola_vilenkin.**BarriolaVilekin**(*c=c*, *k=k*)

Barriola-Vilekin monopol metric Phys. Rev. Lett. 63, 341 Manuel Barriola and Alexander Vilenkin Published 24 July 1989

Parameters

- **c** (*Basic or int or float*) – Any value to assign to speed of light. Defaults to 'c'.
- **k** (*Basic or int or float*) – The scaling factor responsible for the deficit/surplus angle Defaults to k.

Bertotti Kasner

einsteinpy.symbolic.predefined.bertotti_kasner.**BertottiKasner**(*c=c*, *k=k*, *lambd=l*)

Birkhoff's theorem with -term and Bertotti-Kasner space Phys. Lett. A, 245:363–365, 1998 W. Rindler

Parameters

- **c** (*Basic or int or float*) – Any value to assign to speed of light. Defaults to 'c'.
- **lambd** (*Basic or int or float*) – The cosmological constant, note it must be postive. Defaults to l.

Ernst

einsteinpy.symbolic.predefined.ernst.**Ernst**(*B=B*, *M=M*)

Black holes in a magnetic universe. J. Math. Phys., 17:54–56, 1976. Frederick J. Ernst.

Parameters

- **M** (*Basic or int or float*) – Mass of the black hole. Defaults to M.
- **B** (*Basic or int or float*) – The magnetic field strength Defaults to B.

Janis Newman Winicour

einsteinpy.symbolic.predefined.janis_newman_winicour.**JanisNewmanWinicour**(*c=c*, *G=G*, *gam=gam*, *M=M*)

Reality of the Schwarzschild singularity. Phys. Rev. Lett., 20:878–880, 1968. A. I. Janis, E. T. Newman, and J. Winicour.

Parameters

- **M** (*Basic or int or float*) – Mass parameter, this is used for defining the schwarzschild metric. Defaults to M.
- **gam** (*Basic or int or float*) – Parameter for scaling Schwarzschild radius, for gamma=1 this will return the Schwarzschild metric Defaults to gam.

find

This module contains the function find to search for list of possible metrics.

einsteinpy.symbolic.predefined.find.**find**(*search_string*)

Performs a find operation on available functions.

Parameters **search_string** (*str*) – Name of the function to be searched.

Returns A list of available functions related to search_string.

Return type list

Helper Function and Classes

einsteinpy.symbolic.helpers.**simplify_sympy_array**(*arr*)

Function to simplify sympy expression or array.

This function is explicitely defined as native simplify function within sympy stopped working with sympy version change.

Parameters **arr** (*NDimArray or Expr*) – Any sympy array or expression.

Returns Simplified sympy array or expression.

Return type sympy.tensor.array.ndim_array.NDimArray or Expr

einsteinpy.symbolic.helpers.**sympy_to_np_array**(*arr*)

Function to convert sympy to numpy array

Parameters **arr** (*NDimArray*) – Sympy Array

Returns Numpy Array

Return type ndarray

class einsteinpy.symbolic.helpers.**TransformationMatrix**(*iterable*, *old_coords*, *new_coords*, *shape=None*, *old2new=None*, *new2old=None*, ***kwargs*)

Bases: sympy.tensor.array.dense_ndim_array.ImmutableDenseNDimArray

Class for defining transformation matrix for basis change of vectors and tensors.

Constructor.

Parameters

- **iterable** (*iterable-object*) – 2D list or array to pass a matrix.
- **old_coords** (*list or tuple*) – list of old coordinates. For example, `[x, y]`.
- **new_coords** (*list or tuple*) – list of new coordinates. For example, `[r, theta]`.
- **shape** (*tuple, optional*) – shape of the transformation matrix. Usually, not required. Defaults to `None`.
- **old2new** (*list or tuple, optional*) – List of expressions for new coordinates in terms of old coordinates. For example, `[x**2+y**2, atan2(y, x)]`.
- **new2old** (*list or tuple, optional*) – List of expressions for old coordinates in terms of new coordinates. For example, `[r*cos(theta), r*sin(theta)]`.

Raises **ValueError** – Raised when tensor has a rank not equal to 2. This is because, a matrix is expected.

classmethod from_new2old(*old_coords*, *new_coords*, *new2old*, ***kwargs*)

Classmethod to obtain transformation matrix from old coordinates expressed as a function of new coordinates.

Parameters

- **old_coords** (*list or tuple*) – list of old coordinates. For example, `[x, y]`.
- **new_coords** (*list or tuple*) – list of new coordinates. For example, `[r, theta]`.
- **new2old** (*list or tuple, optional*) – List of expressions for old coordinates in terms of new coordinates. For example, `[r*cos(theta), r*sin(theta)]`.

inv()

Returns inverse of the transformation matrix

Returns Inverse of the matrix.

Return type NDimArray

Symbolic Constants Module

This module contains common constants used in physics/relativity and classes used to define them:

class einsteinpy.symbolic.constants.**SymbolicConstant**(*name*, *descriptive_name=None*, ***assumptions*)

Bases: sympy.core.symbol.Symbol

This class inherits from ~sympy.core.symbol.Symbol

Constructor and Initializer

Parameters

- **name** (*str*) – Short, commonly accepted name of the constant. For example, ‘c’ for Speed of light.
- **descriptive_name** (*str*) – The extended name of the constant. For example, ‘Speed of Light’ for ‘c’. Defaults to None.

property descriptive_name
Returns the extended name of the constant

einsteinpy.symbolic.constants.**get_constant**(*name*)
Returns a symbolic instance of the constant

Parameters **name** (*str*) – Name of the constant. Currently available names are 'c', 'G', 'Cosmo_Const', 'eps_0'.

Returns An instance of the required constant

Return type *SymbolicConstant*

Tensor Module

This module contains Tensor class which serves as the base class for more specific Tensors in General Relativity:

einsteinpy.symbolic.tensor.**tensor_product**(*tensor1*, *tensor2*, *i=None*, *j=None*)
Tensor Product of `tensor1` and `tensor2`

Parameters

- **tensor1** (*BaseRelativityTensor*) –
- **tensor2** (*BaseRelativityTensor*) –
- **i** (*int, optional*) – contract i``th index of ``tensor1
- **j** (*int, optional*) – contract j``th index of ``tensor2

Returns tensor of appropriate rank

Return type BaseRelativityTensor

Raises **ValueError** – Raised when `i` and `j` both indicate 'u' or 'l' indices

class einsteinpy.symbolic.tensor.**Tensor**(*arr*, *config='ll'*, *name=None*)
Bases: `object`

Base Class for Tensor manipulation

Constructor and Initializer

Parameters

- **arr** (*ImmutableDenseNDimArray or list*) – Sympy Array, multi-dimensional list containing Sympy Expressions, or Sympy Expressions or int or float scalar
- **config** (*str*) – Configuration of contravariant and covariant indices in tensor. 'u' for upper and 'l' for lower indices. Defaults to 'll'.
- **name** (*str or None*) – Name of the tensor.

Raises

- **TypeError** – Raised when arr is not a list or sympy array
- **TypeError** – Raised when config is not of type str or contains characters other than 'l' or 'u'
- **ValueError** – Raised when `config` implies order of Tensor different than that indicated by shape of `arr`

property order
Returns the order of the Tensor

property config
Returns the configuration of covariant and contravariant indices

tensor()
Returns the sympy Array

Returns Sympy Array object

Return type ImmutableDenseNDimArray

subs(**args*)
Substitute the variables/expressions in a Tensor with other sympy variables/expressions.

Parameters **args** (*one argument or two argument*) –

- two arguments, e.g foo.subs(old, new)
- one iterable argument, e.g foo.subs([(old1, new1), (old2, new2)]) for multiple substitutions at once.

Returns Tensor with substituted values

Return type *Tensor*

simplify(*set_self=True*)
Returns a simplified Tensor

Parameters **set_self** (*bool*) – Replaces the tensor contained the class with its simplified version, if `True`. Defaults to `True`.

Returns Simplified Tensor

Return type *Tensor*

class einsteinpy.symbolic.tensor.**BaseRelativityTensor**(*arr*, *syms*, *config='ll'*, *parent_metric=None*, *variables=[]*, *functions=[]*, *name='GenericTensor'*)
Bases: *einsteinpy.symbolic.tensor.Tensor*

Generic class for defining tensors in General Relativity. This would act as a base class for other Tensorial quantities in GR.

arr
Raw Tensor in sympy array

Type ImmutableDenseNDimArray

syms
List of symbols denoting space and time axis

Type list or tuple

dims
dimension of the space-time.

Type int

variables
free variables in the tensor expression other than the variables describing space-time axis.

Type list

functions
Undefined functions in the tensor expression.

Type list

name
Name of the tensor. Defaults to "GenericTensor".

Type str or None

Constructor and Initializer

Parameters

- **arr** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions
- **syms** (*tuple or list*) – List of crucial symbols dentoting time-axis and/or spacial axis. For example, in case of 4D space-time, the arrangement would look like [t, x1, x2, x3].
- **config** (*str*) – Configuration of contravariant and covariant indices in tensor. 'u' for upper and 'l' for lower indices. Defaults to 'll'.
- **parent_metric** (*MetricTensor or None*) – Metric Tensor for some particular space-time which is associated with this Tensor.
- **variables** (*tuple or list or set*) – List of symbols used in expressing the tensor, other than symbols associated with denoting the space-time axis. Calculates in real-time if left blank.
- **functions** (*tuple or list or set*) – List of symbolic functions used in epressing the tensor. Calculates in real-time if left blank.
- **name** (*str or None*) – Name of the Tensor. Defaults to "GenericTensor".

Raises

- **TypeError** – Raised when arr is not a list, sympy array or numpy array.
- **TypeError** – Raised when config is not of type str or contains characters other than 'l' or 'u'
- **TypeError** – Raised when arguments syms, variables, functions have data type other than list, tuple or set.
- **TypeError** – Raised when argument parent_metric does not belong to MetricTensor class and isn't None.
- **ValueError** – Raised when argument `syms` does not agree with shape of argument `arr`

property parent_metric
Returns the Metric from which Tensor was derived/associated, if available.

symbols()
Returns the symbols used for defining the time & spacial axis

Returns tuple containing (t,x1,x2,x3) in case of 4D space-time

Return type tuple

tensor_lambdify(**args*)
Returns lambdified function of symbolic tensors. This means that the returned functions can accept numerical values and return numerical quantities.

Parameters ***args** – The variable number of arguments accept sympy symbols. The returned function accepts arguments in same order as initially defined in `*args`. Uses sympy symbols from class attributes `syms` and `variables` (in the same order) if no `*args` is passed Leaving `*args` empty is recommended.

Returns

- *tuple* – arguments to be passed in the returned function in exact order.
- *function* – Lambdified function which accepts and returns numerical quantities.

lorentz_transform(*transformation_matrix*)
Performs a Lorentz transform on the tensor.

Parameters **transformation_matrix** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions

Returns lorentz transformed tensor(or vector)

Return type *BaseRelativityTensor*

Vector module

This module contains the class GenericVector to represent a vector in arbitrary space-time symbolically

class einsteinpy.symbolic.vector.**GenericVector**(*arr*, *syms*, *config='u'*, *parent_metric=None*, *name='GenericVector'*)
Bases: *einsteinpy.symbolic.tensor.BaseRelativityTensor*

Class to represent a vector in arbitrary space-time symbolically

Constructor and Initializer

Parameters

- **arr** (*ImmutableDenseNDimArray*) – Sympy Array containing Sympy Expressions
- **syms** (*tuple or list*) – Tuple of crucial symbols denoting time-axis, 1st, 2nd, and 3rd axis (t,x1,x2,x3)
- **config** (*str*) – Configuration of contravariant and covariant indices in tensor. 'u' for upper and 'l' for lower indices. Defaults to 'u'.
- **parent_metric** (MetricTensor *or None*) – Corresponding Metric for the Generic Vector. Defaults to None.
- **name** (*str*) – Name of the Vector. Defaults to "GenericVector".

Raises

- **ValueError** – config has more than 1 index
- **ValueError** – Dimension should be equal to 1

change_config(*newconfig='u'*, *metric=None*)
Changes the index configuration(contravariant/covariant)

Parameters

- **newconfig** (*str*) – Specify the new configuration. Defaults to 'u'
- **metric** (MetricTensor *or None*) – Parent metric tensor for changing indices. Already assumes the value of the metric tensor from which it was initialized if passed with None. Defaults to None.

Returns New tensor with new configuration.

Return type *GenericVector*

Raises **Exception** – Raised when a parent metric could not be found.

lorentz_transform(*transformation_matrix*)
Performs a Lorentz transform on the vector.

Parameters **transformation_matrix** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions

Returns lorentz transformed vector

Return type *GenericVector*

Metric Tensor Module

This module contains the class for defining a Metric of an arbitrary spacetime, symbolically. Note that, only the coordinate symbols are to be supplied to the `syms` parameter, while `arr` takes the metric (as a SymPy array), which may contain several constants. The symbols in `syms` define the basis to perform several operations on the metric, such as symbolic differentiation. Symbols, for the constants in the metric, should be defined independently and used directly in the specification of `arr`. Please check the metric definitions in `einsteinpy.symbolic.predefined` for examples of doing this.

class einsteinpy.symbolic.metric.**MetricTensor**(*arr*, *syms*, *config='ll'*, *name='GenericMetricTensor'*)
Bases: *einsteinpy.symbolic.tensor.BaseRelativityTensor*

Class to define a metric tensor for a space-time

Constructor and Initializer

Parameters

- **arr** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions
- **syms** (*tuple or list*) – Tuple of crucial symbols denoting time-axis, 1st, 2nd, and 3rd axis (t,x1,x2,x3)
- **config** (*str*) – Configuration of contravariant and covariant indices in tensor. 'u' for upper and 'l' for lower indices. Defaults to 'll'.
- **name** (*str*) – Name of the Metric Tensor. Defaults to "GenericMetricTensor".

Raises

- **TypeError** – Raised when arr is not a list or sympy Array
- **TypeError** – syms is not a list or tuple
- **ValueError** – config has more or less than 2 indices

change_config(*newconfig='uu'*)
Changes the index configuration(contravariant/covariant)

Parameters **newconfig** (*str*) – Specify the new configuration. Defaults to 'uu'

Returns New Metric with new configuration. Defaults to 'uu'

Return type *MetricTensor*

Raises **ValueError** – Raised when new configuration is not 'll' or 'uu'. This constraint is in place because we are dealing with Metric Tensor.

inv()
Returns the inverse of the Metric. Returns contravariant Metric if it is originally covariant or vice-versa.

Returns New Metric which is the inverse of original Metric.

Return type *MetricTensor*

lower_config()
Returns a covariant instance of the given metric tensor.

Returns same instance if the configuration is already lower or inverse of given metric if configuration is upper

Return type *MetricTensor*

lorentz_transform(*transformation_matrix*)
Performs a Lorentz transform on the tensor.

Parameters **transformation_matrix** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions

Returns lorentz transformed tensor

Return type *MetricTensor*

Christoffel Symbols Module

This module contains the class for obtaining Christoffel Symbols related to a Metric belonging to any arbitrary space-time symbolically:

class einsteinpy.symbolic.christoffel.**ChristoffelSymbols**(*arr*, *syms*, *config='ull'*, *parent_metric=None*, *name='ChristoffelSymbols'*)
Bases: *einsteinpy.symbolic.tensor.BaseRelativityTensor*

Class for defining christoffel symbols.

Constructor and Initializer

Parameters

- **arr** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions
- **syms** (*tuple or list*) – Tuple of crucial symbols denoting time-axis, 1st, 2nd, and 3rd axis (t,x1,x2,x3)
- **config** (*str*) – Configuration of contravariant and covariant indices in tensor. 'u' for upper and 'l' for lower indices. Defaults to 'ull'.
- **parent_metric** (MetricTensor) – Metric Tensor from which Christoffel symbol is calculated. Defaults to None.
- **name** (*str*) – Name of the Christoffel Symbols Tensor. Defaults to "ChristoffelSymbols".

Raises

- **TypeError** – Raised when arr is not a list or sympy Array
- **TypeError** – syms is not a list or tuple
- **ValueError** – config has more or less than 3 indices

classmethod from_metric(*metric*)
Get Christoffel symbols calculated from a metric tensor

Parameters **metric** (MetricTensor) – Space-time Metric from which Christoffel Symbols are to be calculated

change_config(*newconfig='lll'*, *metric=None*)
Changes the index configuration(contravariant/covariant)

Parameters

- **newconfig** (*str*) – Specify the new configuration. Defaults to 'lll'
- **metric** (*MetricTensor or None*) – Parent metric tensor for changing indices. Already assumes the value of the metric tensor from which it was initialized if passed with None. Compulsory if not initialized with 'from_metric'. Defaults to None.

Returns New tensor with new configuration. Defaults to 'lll'

Return type *ChristoffelSymbols*

Raises **Exception** – Raised when a parent metric could not be found.

lorentz_transform(*transformation_matrix*)
Performs a Lorentz transform on the tensor.

Parameters **transformation_matrix** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions

Returns lorentz transformed tensor

Return type *ChristoffelSymbols*

Riemann Tensor Module

This module contains the class for obtaining Riemann Curvature Tensor related to a Metric belonging to any arbitrary space-time symbolically:

class einsteinpy.symbolic.riemann.**RiemannCurvatureTensor**(*arr*, *syms*, *config='ulll'*, *parent_metric=None*, *name='RiemannCurvatureTensor'*)
Bases: *einsteinpy.symbolic.tensor.BaseRelativityTensor*

Class for defining Riemann Curvature Tensor

Constructor and Initializer

Parameters

- **arr** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions
- **syms** (*tuple or list*) – Tuple of crucial symbols denoting time-axis, 1st, 2nd, and 3rd axis (t,x1,x2,x3)
- **config** (*str*) – Configuration of contravariant and covariant indices in tensor. 'u' for upper and 'l' for lower indices. Defaults to 'ulll'.
- **parent_metric** (*MetricTensor*) – Metric Tensor related to this Riemann Curvature Tensor.
- **name** (*str*) – Name of the Tensor. Defaults to "RiemannCurvatureTensor".

Raises

- **TypeError** – Raised when arr is not a list or sympy Array
- **TypeError** – syms is not a list or tuple
- **ValueError** – config has more or less than 4 indices

classmethod from_christoffels(*chris*, *parent_metric=None*)
Get Riemann Tensor calculated from a Christoffel Symbols

Parameters

- **chris** (ChristoffelSymbols) – Christoffel Symbols from which Riemann Curvature Tensor to be calculated
- **parent_metric** (MetricTensor *or* None) – Corresponding Metric for the Riemann Tensor. None if it should inherit the Parent Metric of Christoffel Symbols. Defaults to None.

classmethod from_metric(*metric*)
Get Riemann Tensor calculated from a Metric Tensor

Parameters **metric** (MetricTensor) – Metric Tensor from which Riemann Curvature Tensor to be calculated

change_config(*newconfig='llll'*, *metric=None*)
Changes the index configuration(contravariant/covariant)

Parameters

- **newconfig** (*str*) – Specify the new configuration. Defaults to 'llll'
- **metric** (MetricTensor *or* None) – Parent metric tensor for changing indices. Already assumes the value of the metric tensor from which it was initialized if passed with None. Compulsory if not initialized with 'from_metric'. Defaults to None.

Returns New tensor with new configuration. Configuration defaults to 'llll'

Return type *RiemannCurvatureTensor*

Raises **Exception** – Raised when a parent metric could not be found.

lorentz_transform(*transformation_matrix*)
Performs a Lorentz transform on the tensor.

Parameters **transformation_matrix** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions

Returns lorentz transformed tensor

Return type *RiemannCurvatureTensor*

Ricci Module

This module contains the basic classes for obtaining Ricci Tensor and Ricci Scalar related to a Metric belonging to any arbitrary space-time symbolically:

class einsteinpy.symbolic.ricci.**RicciTensor**(*arr*, *syms*, *config='ll'*, *parent_metric=None*, *name='RicciTensor'*)
Bases: *einsteinpy.symbolic.tensor.BaseRelativityTensor*

Class for defining Ricci Tensor

Constructor and Initializer

Parameters

- **arr** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions

- **syms** (*tuple or list*) – Tuple of crucial symbols denoting time-axis, 1st, 2nd, and 3rd axis (t,x1,x2,x3)
- **config** (*str*) – Configuration of contravariant and covariant indices in tensor. 'u' for upper and 'l' for lower indices. Defaults to 'll'.
- **parent_metric** (MetricTensor *or None*) – Corresponding Metric for the Ricci Tensor. Defaults to None.
- **name** (*str*) – Name of the Tensor. Defaults to "RicciTensor".

Raises

- **TypeError** – Raised when arr is not a list or sympy Array
- **TypeError** – syms is not a list or tuple
- **ValueError** – config has more or less than 2 indices

classmethod from_riemann(*riemann*, *parent_metric=None*)
Get Ricci Tensor calculated from Riemann Tensor

Parameters

- **riemann** (RiemannCurvatureTensor) – Riemann Tensor
- **parent_metric** (MetricTensor *or None*) – Corresponding Metric for the Ricci Tensor. None if it should inherit the Parent Metric of Riemann Tensor. Defaults to None.

classmethod from_christoffels(*chris*, *parent_metric=None*)
Get Ricci Tensor calculated from Christoffel Tensor

Parameters

- **chris** (ChristoffelSymbols) – Christoffel Tensor
- **parent_metric** (MetricTensor *or None*) – Corresponding Metric for the Ricci Tensor. None if it should inherit the Parent Metric of Christoffel Symbols. Defaults to None.

classmethod from_metric(*metric*)
Get Ricci Tensor calculated from Metric Tensor

Parameters **metric** (MetricTensor) – Metric Tensor

change_config(*newconfig='ul'*, *metric=None*)
Changes the index configuration(contravariant/covariant)

Parameters

- **newconfig** (*str*) – Specify the new configuration. Defaults to 'ul'
- **metric** (MetricTensor *or None*) – Parent metric tensor for changing indices. Already assumes the value of the metric tensor from which it was initialized if passed with None. Compulsory if somehow does not have a parent metric. Defaults to None.

Returns New tensor with new configuration. Defaults to 'ul'

Return type *RicciTensor*

Raises **Exception** – Raised when a parent metric could not be found.

lorentz_transform(*transformation_matrix*)
Performs a Lorentz transform on the tensor.

Parameters **transformation_matrix** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions

Returns lorentz transformed tensor

Return type *RicciTensor*

class einsteinpy.symbolic.ricci.**RicciScalar**(*arr*, *syms*, *parent_metric=None*)

Bases: *einsteinpy.symbolic.tensor.BaseRelativityTensor*

Class for defining Ricci Scalar

Constructor and Initializer

Parameters

- **arr** (*ImmutableDenseNDimArray or list*) – Sympy Array, multi-dimensional list containing Sympy Expressions, or Sympy Expressions or int or float scalar
- **syms** (*tuple or list*) – Tuple of crucial symbols denoting time-axis, 1st, 2nd, and 3rd axis (t,x1,x2,x3)
- **parent_metric** (MetricTensor *or None*) – Corresponding Metric for the Ricci Scalar. Defaults to None.

Raises **TypeError** – Raised when syms is not a list or tuple

property expr

Retuns the symbolic expression of the Ricci Scalar

classmethod from_riccitensor(*riccitensor*, *parent_metric=None*)

Get Ricci Scalar calculated from Ricci Tensor

Parameters

- **riccitensor** (*RicciTensor*) – Ricci Tensor
- **parent_metric** (MetricTensor *or None*) – Corresponding Metric for the Ricci Scalar. Defaults to None.

classmethod from_riemann(*riemann*, *parent_metric=None*)

Get Ricci Scalar calculated from Riemann Tensor

Parameters

- **riemann** (RiemannCurvatureTensor) – Riemann Tensor
- **parent_metric** (MetricTensor *or None*) – Corresponding Metric for the Ricci Scalar. Defaults to None.

classmethod from_christoffels(*chris*, *parent_metric=None*)

Get Ricci Scalar calculated from Christoffel Tensor

Parameters

- **chris** (ChristoffelSymbols) – Christoffel Tensor
- **parent_metric** (MetricTensor *or None*) – Corresponding Metric for the Ricci Scalar. Defaults to None.

classmethod from_metric(*metric*)

Get Ricci Scalar calculated from Metric Tensor

Parameters **metric** (MetricTensor) – Metric Tensor

Einstein Tensor Module

This module contains the class for obtaining Einstein Tensor related to a Metric belonging to any arbitrary space-time symbolically:

class einsteinpy.symbolic.einstein.**EinsteinTensor**(*arr*, *syms*, *config='ll'*, *parent_metric=None*, *name='EinsteinTensor'*)

Bases: *einsteinpy.symbolic.tensor.BaseRelativityTensor*

Class for defining Einstein Tensor

Constructor and Initializer

Parameters

- **arr** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions
- **syms** (*tuple or list*) – Tuple of crucial symbols denoting time-axis, 1st, 2nd, and 3rd axis (t,x1,x2,x3)
- **config** (*str*) – Configuration of contravariant and covariant indices in tensor. 'u' for upper and 'l' for lower indices. Defaults to 'll'.
- **parent_metric** (*MetricTensor or None*) – Corresponding Metric for the Einstein Tensor. Defaults to None.
- **name** (*str*) – Name of the Tensor. Defaults to "EinsteinTensor".

Raises

- **TypeError** – Raised when arr is not a list or sympy Array
- **TypeError** – syms is not a list or tuple
- **ValueError** – config has more or less than 2 indices

change_config(*newconfig='ul'*, *metric=None*)

Changes the index configuration(contravariant/covariant)

Parameters

- **newconfig** (*str*) – Specify the new configuration. Defaults to 'ul'
- **metric** (*MetricTensor or None*) – Parent metric tensor for changing indices. Already assumes the value of the metric tensor from which it was initialized if passed with None. Compulsory if somehow does not have a parent metric. Defaults to None.

Returns New tensor with new configuration. Defaults to 'ul'

Return type *EinsteinTensor*

Raises **Exception** – Raised when a parent metric could not be found.

lorentz_transform(*transformation_matrix*)

Performs a Lorentz transform on the tensor.

Parameters **transformation_matrix** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions

Returns lorentz transformed tensor

Return type *EinsteinTensor*

Stress Energy Momentum Tensor Module

This module contains the class for obtaining Stress Energy Momentum Tensor related to a Metric belonging to any arbitrary space-time symbolically:

class einsteinpy.symbolic.stress_energy_momentum.**StressEnergyMomentumTensor**(*arr*, *syms*, *config='ll'*, *parent_metric=None*, *name='StressEnergyM*

Bases: *einsteinpy.symbolic.tensor.BaseRelativityTensor*

Class for defining Stress-Energy Coefficient Tensor

Constructor and Initializer

Parameters

- **arr** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions
- **syms** (*tuple or list*) – Tuple of crucial symbols denoting time-axis, 1st, 2nd, and 3rd axis (t,x1,x2,x3)
- **config** (*str*) – Configuration of contravariant and covariant indices in tensor. 'u' for upper and 'l' for lower indices. Defaults to 'll'.
- **parent_metric** (MetricTensor *or None*) – Corresponding Metric for the Stress-Energy Coefficient Tensor. Defaults to None.
- **name** (*str*) – Name of the Tensor. Defaults to "StressEnergyMomentumTensor".

Raises

- **TypeError** – Raised when arr is not a list or sympy Array
- **TypeError** – syms is not a list or tuple
- **ValueError** – config has more or less than 2 indices

change_config(*newconfig='ul'*, *metric=None*)

Changes the index configuration(contravariant/covariant)

Parameters

- **newconfig** (*str*) – Specify the new configuration. Defaults to 'ul'
- **metric** (MetricTensor *or None*) – Parent metric tensor for changing indices. Already assumes the value of the metric tensor from which it was initialized if passed with None. Compulsory if somehow does not have a parent metric. Defaults to None.

Returns New tensor with new configuration. Defaults to 'ul'

Return type *StressEnergyMomentumTensor*

Raises **Exception** – Raised when a parent metric could not be found.

lorentz_transform(*transformation_matrix*)

Performs a Lorentz transform on the tensor.

Parameters **transformation_matrix** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions

Returns lorentz transformed tensor

Return type *StressEnergyMomentumTensor*

Weyl Tensor Module

This module contains the class for obtaining Weyl Tensor related to a Metric belonging to any arbitrary space-time symbolically:

class einsteinpy.symbolic.weyl.**WeylTensor**(*arr*, *syms*, *config='ulll'*, *parent_metric=None*, *name='WeylTensor'*)

Bases: *einsteinpy.symbolic.tensor.BaseRelativityTensor*

Class for defining Weyl Tensor

Constructor and Initializer

Parameters

- **arr** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions
- **syms** (*tuple or list*) – Tuple of crucial symbols denoting time-axis, 1st, 2nd, and 3rd axis (t,x1,x2,x3)
- **config** (*str*) – Configuration of contravariant and covariant indices in tensor. 'u' for upper and 'l' for lower indices. Defaults to 'ulll'.
- **parent_metric** (*WeylTensor*) – Corresponding Metric for the Weyl Tensor. Defaults to None.
- **name** (*str*) – Name of the Tensor. Defaults to "WeylTensor"

Raises

- **TypeError** – Raised when arr is not a list or sympy Array
- **TypeError** – syms is not a list or tuple
- **ValueError** – config has more or less than 4 indices

classmethod from_metric(*metric*)

Get Weyl tensor calculated from a metric tensor

Parameters **metric** (MetricTensor) – Space-time Metric from which Christoffel Symbols are to be calculated

Raises **ValueError** – Raised when the dimension of the tensor is less than 3

change_config(*newconfig='llll'*, *metric=None*)

Changes the index configuration(contravariant/covariant)

Parameters

- **newconfig** (*str*) – Specify the new configuration. Defaults to 'llll'
- **metric** (MetricTensor *or None*) – Parent metric tensor for changing indices. Already assumes the value of the metric tensor from which it was initialized if passed with None. Compulsory if not initialized with 'from_metric'. Defaults to None.

Returns New tensor with new configuration. Configuration defaults to 'llll'

Return type *WeylTensor*

Raises **Exception** – Raised when a parent metric could not be found.

lorentz_transform(*transformation_matrix*)
Performs a Lorentz transform on the tensor.

Parameters **transformation_matrix** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions

Returns lorentz transformed tensor(or vector)

Return type *WeylTensor*

Schouten Module

This module contains the basic classes for obtaining Schouten Tensor related to a Metric belonging to any arbitrary space-time symbolically:

class einsteinpy.symbolic.schouten.**SchoutenTensor**(*arr*, *syms*, *config='ll'*, *parent_metric=None*, *name='SchoutenTensor'*)
Bases: *einsteinpy.symbolic.tensor.BaseRelativityTensor*

Class for defining Schouten Tensor

Constructor and Initializer

Parameters

- **arr** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions
- **syms** (*tuple or list*) – Tuple of crucial symbols denoting time-axis, 1st, 2nd, and 3rd axis (t,x1,x2,x3)
- **config** (*str*) – Configuration of contravariant and covariant indices in tensor. 'u' for upper and 'l' for lower indices. Defaults to 'll'.
- **parent_metric** (MetricTensor) – Corresponding Metric for the Schouten Tensor. Defaults to None.
- **name** (*str*) – Name of the Tensor. Defaults to "SchoutenTensor".

Raises

- **TypeError** – Raised when arr is not a list or sympy Array
- **TypeError** – syms is not a list or tuple
- **ValueError** – config has more or less than 2 indices

classmethod from_metric(*metric*)
Get Schouten tensor calculated from a metric tensor

Parameters **metric** (MetricTensor) – Space-time Metric from which Christoffel Symbols are to be calculated

Raises **ValueError** – Raised when the dimension of the tensor is less than 3

change_config(*newconfig='ul'*, *metric=None*)
Changes the index configuration(contravariant/covariant)

Parameters

- **newconfig** (*str*) – Specify the new configuration. Defaults to 'ul'

- **metric** (*MetricTensor or None*) – Parent metric tensor for changing indices. Already assumes the value of the metric tensor from which it was initialized if passed with None. Compulsory if not initialized with 'from_metric'. Defaults to None.

Returns New tensor with new configuration. Configuration defaults to 'ul'

Return type *SchoutenTensor*

Raises **Exception** – Raised when a parent metric could not be found.

lorentz_transform(*transformation_matrix*)
Performs a Lorentz transform on the tensor.

Parameters **transformation_matrix** (*ImmutableDenseNDimArray or list*) – Sympy Array or multi-dimensional list containing Sympy Expressions

Returns lorentz transformed tensor

Return type *SchoutenTensor*

1.7.4 Hypersurface module

This module contains Classes to calculate and plot hypersurfaces of various geometries.

Schwarzschild Embedding Module

Class for Utility functions for Schwarzschild Embedding surface to implement gravitational lensing:

class einsteinpy.hypersurface.schwarzschildembedding.**SchwarzschildEmbedding**(*M*)
Bases: object

Class for Utility functions for Schwarzschild Embedding surface to implement gravitational lensing

input_units
list of input units of M

Type list

units_list
customized units to handle values of M and render plots within grid range

Type list

r_init

Type m

Constructor Initialize mass and embedding initial radial coordinate in appropiate units in order to render the plots of the surface in finite grid. The initial r is taken to be just greater than schwarzschild radius but it is important to note that the embedding breaks at r < 9m/4.

Parameters **M** (*kg*) – Mass of the body

gradient(*r*)
Calculate gradient of Z coordinate w.r.t r to update the value of r and thereby get value of spherical radial coordinate R.

Parameters **r** (*float*) – schwarzschild coordinate at which gradient is supposed to be obtained

Returns gradient of Z w.r.t r at the point r (passed as argument)

Return type float

radial_coord(*r*)
Returns spherical radial coordinate (of the embedding) from given schwarzschild coordinate.

Parameters **r** (*float*) –

Returns spherical radial coordinate of the 3d embedding

Return type float

get_values(*alpha*)
Obtain the Z coordinate values and corrosponding R values for range of r as 9m/4 < r < 9m.

Parameters **alpha** (*float*) – scaling factor to obtain the step size for incrementing r

Returns (list, list) : values of R (x_axis) and Z (y_axis)

Return type tuple

get_values_surface(*alpha*)
Obtain the same values as of the get_values function but reshapes them to obtain values for all points on the solid of revolution about Z axis (as the embedding is symmetric in angular coordinates).

Parameters **alpha** (*float*) – scaling factor to obtain the step size for incrementing r

Returns (~numpy.array of X, ~numpy.array of Y, ~numpy.array of Z) values in cartesian coordinates obtained after applying solid of revolution

Return type tuple

1.7.5 Rays module

This module contains Classes to calculate and plot trajectories and other interactions of light with heavy objects.

Shadow Module

Module for calculating Shadow cast by an thin emission disk around schwarzschild spacetime.

class einsteinpy.rays.shadow.**Shadow**(*mass*, *n_rays*, *fov*, *limit=0.001*)
Bases: object

Class for plotting the shadow of Schwarzschild Black Hole surrounded by a thin accreting emission disk as seen by a distant observer.

smoothen(*points=500*)
Sets the interpolated values for the intensities for smoothening of the plot using ~scipy.interpolate.interp1d

1.7.6 Utils module

The common utilities of the project are included in this module.

Scalar Factor

einsteinpy.utils.scalar_factor.**scalar_factor**(*t*, *era='md'*, *tuning_param=1.0*)

Acceleration of the universe in cosmological models of Robertson Walker Flat Universe.

Parameters

- **era** (*string*) – Can be chosen from 'md' (Matter Dominant), 'rd' (Radiation Dominant) and 'ded' (Dark Energy Dominant)
- **t** (*s*) – Time for the event
- **tuning_param** (*float, optional*) – Unit scaling factor, defaults to 1

Returns Value of scalar factor at time t.

Return type float

:raises ValueError : If era is not 'md' , 'rd', and 'ded'.:

einsteinpy.utils.scalar_factor.**scalar_factor_derivative**(*t*, *era='md'*, *tuning_param=1.0*)

Derivative of acceleration of the universe in cosmological models of Robertson Walker Flat Universe.

Parameters

- **era** (*string*) – Can be chosen from 'md' (Matter Dominant), 'rd' (Radiation Dominant) and 'ded' (Dark Energy Dominant)
- **t** (*s*) – Time for the event
- **tuning_param** (*float, optional*) – Unit scaling factor, defaults to 1

Returns Value of derivative of scalar factor at time t.

Return type float

:raises ValueError : If era is not 'md' , 'rd', and 'ded'.:

Exceptions module

Docstring for exceptions.py module

This module defines the `BaseError` class which is the base class for all custom Errors in EinsteinPy, and the `CoordinateError` class, which is a child class used for raising exceptions when the geometry does not support the supplied coordinate system.

exception einsteinpy.utils.exceptions.**BaseError**(**args*, ***kwargs*)

Base class for custom errors

Constructor

Joins `args` into a `message` string

Parameters

- ***args** (*iterable*) – Other arguments
- ****kwargs** (*dict*) – Keyword arguments

exception einsteinpy.utils.exceptions.**CoordinateError**(**args*, ***kwargs*)

Error class for invalid coordinate operations

Constructor
Joins `args` into a `message` string

Parameters

- ***args** (*iterable*) – Other arguments
- ****kwargs** (*dict*) – Keyword arguments

1.7.7 Plotting module

This module contains the basic classes for static and interactive 3-D and 2-D geodesic, hypersurface and shadow plotting modules.

Geodesics Plotting module

This module contains Classes to plot geodesics.

Core plotting module

This module contains the advanced backend switching while plotting.

class einsteinpy.plotting.geodesic.core.**GeodesicPlotter**(*ax=None*, *bh_colors=('#000', '#FFC')*, *draw_ergosphere=True*)

Bases: `einsteinpy.plotting.geodesic.static.StaticGeodesicPlotter`

Class for automatically switching between Matplotlib and Plotly depending on platform used.

Constructor

Parameters

- **ax** (*Axes*) – Matplotlib Axes object To be deprecated in Version 0.5.0 Since Version 0.4.0, *StaticGeodesicPlotter* automatically creates a new Axes Object. Defaults to `None`
- **bh_colors** (*tuple, optional*) – 2-Tuple, containing hexcodes (Strings) for the colors, used for the Black Hole Event Horizon (Outer) and Ergosphere (Outer) Defaults to `("#000", "#FFC")`
- **draw_ergosphere** (*bool, optional*) – Whether to draw the ergosphere Defaults to *True*

Interactive plotting module

This module contains interactive plotting backend.

Hypersurface plotting module

This module contains the hypersurface plotting.

class einsteinpy.plotting.hypersurface.core.**HypersurfacePlotter**(*embedding*, *plot_type='wireframe'*, *alpha=100*)

Bases: object

Class for plotting and visualising hypersurfaces

Constructor for plotter.

Parameters

- **embedding** (*SchwarzschildEmbedding*) – The embedding of the hypersurface.
- **plot_type** (*str, optional*) – type of texture for the plots - wireframe / surface, defaults to 'wireframe'
- **alpha** (*float, optional*) – scaling factor to obtain the step size for incrementing r, defaults to 100

plot()
Plots the surface thus obtained for the embedding.

show()
Shows the plot.

Rays Plotting module

This module contains classes and sub-modules for light interactions with heavy bodies.

Shadow plotting module

This module contains the black hole shadow plotting class.

class einsteinpy.plotting.rays.shadow.**ShadowPlotter**(*shadow*, *is_line_plot=True*)

Bases: object

Class for plotting and visualising shadows

Constructor for plotter.

Parameters

- **shadow** (*Shadow*) – The shadow object
- **is_line_plot** (*bool, optional*) – If the plot is a line plot or a contour plot. Defaults to True.

plot()
Plots the shadow.

show()
Shows the plot.

1.7.8 Coordinates module

This module contains the classes for various coordinate systems and their position and velocity transformations.

core module

This module contains the basic classes for coordinate systems and their position transformation:

class einsteinpy.coordinates.core.**Cartesian**(*t*, *x*, *y*, *z*)

Bases: einsteinpy.coordinates.conversion.CartesianConversion

Class for defining 3-Position & 4-Position in Cartesian Coordinates using SI units

Constructor

Parameters

- **t** (*float*) – Time
- **x** (*float*) – x-Component of 3-Position
- **y** (*float*) – y-Component of 3-Position
- **z** (*float*) – z-Component of 3-Position

position()

Returns Position 4-Vector in SI units

Returns 4-Tuple, containing Position 4-Vector in SI units

Return type tuple

to_spherical(***kwargs*)

Method for conversion to Spherical Polar Coordinates

Other Parameters ****kwargs** (*dict*) – Keyword Arguments

Returns Spherical representation of the Cartesian Coordinates

Return type *Spherical*

to_bl(***kwargs*)

Method for conversion to Boyer-Lindquist (BL) Coordinates

Parameters ****kwargs** (*dict*) – Keyword Arguments Expects two arguments, `M and ``a`, as described below

Other Parameters

- **M** (*float*) – Mass of gravitating body Required to calculate `alpha`, the rotational length parameter
- **a** (*float*) – Spin Parameter of gravitating body 0 <= a <= 1 Required to calculate `alpha`, the rotational length parameter

Returns Boyer-Lindquist representation of the Cartesian Coordinates

Return type *BoyerLindquist*

class einsteinpy.coordinates.core.**Spherical**(*t*, *r*, *theta*, *phi*)

Bases: einsteinpy.coordinates.conversion.SphericalConversion

Class for defining 3-Position & 4-Position in Spherical Polar Coordinates using SI units

Constructor

Parameters

- **t** (*float*) – Time
- **r** (*float*) – r-Component of 3-Position
- **theta** (*float*) – theta-Component of 3-Position
- **phi** (*float*) – phi-Component of 3-Position

position()
Returns Position 4-Vector in SI units

Returns 4-Tuple, containing Position 4-Vector in SI units

Return type tuple

to_cartesian(***kwargs*)
Method for conversion to Cartesian Coordinates

Other Parameters ****kwargs** (*dict*) – Keyword Arguments

Returns Cartesian representation of the Spherical Polar Coordinates

Return type *Cartesian*

to_bl(***kwargs*)
Method for conversion to Boyer-Lindquist (BL) Coordinates

Parameters ****kwargs** (*dict*) – Keyword Arguments Expects two arguments, `M and ``a`, as described below

Other Parameters

- **M** (*float*) – Mass of gravitating body Required to calculate `alpha`, the rotational length parameter
- **a** (*float*) – Spin Parameter of gravitating body 0 <= a <= 1 Required to calculate `alpha`, the rotational length parameter

Returns Boyer-Lindquist representation of the Spherical Polar Coordinates

Return type *BoyerLindquist*

class einsteinpy.coordinates.core.**BoyerLindquist**(*t*, *r*, *theta*, *phi*)
Bases: einsteinpy.coordinates.conversion.BoyerLindquistConversion

Class for defining 3-Position & 4-Position in Boyer-Lindquist Coordinates using SI units

Constructor

Parameters

- **t** (*float*) – Time
- **r** (*float*) – r-Component of 3-Position
- **theta** (*float*) – theta-Component of 3-Position
- **phi** (*float*) – phi-Component of 3-Position

position()
Returns Position 4-Vector in SI units

Returns 4-Tuple, containing Position 4-Vector in SI units

Return type tuple

to_cartesian(***kwargs*)
Method for conversion to Cartesian Coordinates

Parameters ****kwargs** (*dict*) – Keyword Arguments Expects two arguments, `M and ``a`, as described below

Other Parameters

- **M** (*float*) – Mass of gravitating body Required to calculate `alpha`, the rotational length parameter
- **a** (*float*) – Spin Parameter of gravitating body 0 <= a <= 1 Required to calculate `alpha`, the rotational length parameter

Returns Cartesian representation of the Boyer-Lindquist Coordinates

Return type *Cartesian*

to_spherical(***kwargs*)
Method for conversion to Spherical Polar Coordinates

Parameters ****kwargs** (*dict*) – Keyword Arguments Expects two arguments, `M and ``a`, as described below

Other Parameters

- **M** (*float*) – Mass of gravitating body Required to calculate `alpha`, the rotational length parameter
- **a** (*float*) – Spin Parameter of gravitating body 0 <= a <= 1 Required to calculate `alpha`, the rotational length parameter

Returns Spherical Polar representation of the Boyer-Lindquist Coordinates

Return type *Spherical*

differential module

This module contains the basic classes for time differentials of coordinate systems and the transformations:

class einsteinpy.coordinates.differential.**CartesianDifferential**(*t*, *x*, *y*, *z*, *v_x*, *v_y*, *v_z*)

Bases: `einsteinpy.coordinates.conversion.CartesianConversion`

Class for defining 3-Velocity & 4-Velocity in Cartesian Coordinates using SI units

Constructor

Parameters

- **t** (*Quantity*) – Time
- **x** (*Quantity*) – x-Component of 3-Position
- **y** (*Quantity*) – y-Component of 3-Position
- **z** (*Quantity*) – z-Component of 3-Position
- **v_x** (*Quantity, optional*) – x-Component of 3-Velocity
- **v_y** (*Quantity, optional*) – y-Component of 3-Velocity
- **v_z** (*Quantity, optional*) – z-Component of 3-Velocity

position()
Returns Position 4-Vector in SI units

Returns 4-Tuple, containing Position 4-Vector in SI units

Return type tuple

property v_t
Returns the Timelike component of 4-Velocity

velocity(*metric*)
Returns Velocity 4-Vector in SI units

Parameters **metric** (*) – Metric object, in which the coordinates are defined

Returns 4-Tuple, containing Velocity 4-Vector in SI units

Return type tuple

spherical_differential(**kwargs)
Converts to Spherical Polar Coordinates

Parameters ****kwargs** (*dict*) – Keyword Arguments

Returns Spherical Polar representation of velocity

Return type *SphericalDifferential*

bl_differential(**kwargs)
Converts to Boyer-Lindquist Coordinates

Parameters ****kwargs** (*dict*) – Keyword Arguments Expects two arguments, `M and ``a, as described below

Other Parameters

- **M** (*float*) – Mass of the gravitating body, around which, spacetime has been defined
- **a** (*float*) – Spin Parameter of the gravitating body, around which, spacetime has been defined

Returns Boyer-Lindquist representation of velocity

Return type *BoyerLindquistDifferential*

class einsteinpy.coordinates.differential.**SphericalDifferential**(*t*, *r*, *theta*, *phi*, *v_r*, *v_th*, *v_p*)

Bases: einsteinpy.coordinates.conversion.SphericalConversion

Class for defining 3-Velocity & 4-Velocity in Spherical Polar Coordinates using SI units

Constructor

Parameters

- **t** (*float*) – Time
- **r** (*float*) – r-Component of 3-Position
- **theta** (*float*) – theta-Component of 3-Position
- **phi** (*float*) – phi-Component of 3-Position
- **v_r** (*float, optional*) – r-Component of 3-Velocity
- **v_th** (*float, optional*) – theta-Component of 3-Velocity
- **v_p** (*float, optional*) – phi-Component of 3-Velocity

position()
Returns Position 4-Vector in SI units

Returns 4-Tuple, containing Position 4-Vector in SI units

Return type tuple

property v_t
Returns the Timelike component of 4-Velocity

velocity(*metric*)
Returns Velocity 4-Vector in SI units

Parameters **metric** (*) – Metric object, in which the coordinates are defined

Returns 4-Tuple, containing Velocity 4-Vector in SI units

Return type tuple

cartesian_differential(**kwargs)
Converts to Cartesian Coordinates

Parameters ****kwargs** (*dict*) – Keyword Arguments

Returns Cartesian representation of velocity

Return type *CartesianDifferential*

bl_differential(**kwargs)
Converts to Boyer-Lindquist coordinates

Parameters ****kwargs** (*dict*) – Keyword Arguments Expects two arguments, `M and ``a, as described below

Other Parameters

- **M** (*float*) – Mass of the gravitating body, around which, spacetime has been defined
- **a** (*float*) – Spin Parameter of the gravitating body, around which, spacetime has been defined

Returns Boyer-Lindquist representation of velocity

Return type *BoyerLindquistDifferential*

class einsteinpy.coordinates.differential.**BoyerLindquistDifferential**(*t*, *r*, *theta*, *phi*, *v_r*, *v_th*, *v_p*)

Bases: einsteinpy.coordinates.conversion.BoyerLindquistConversion

Class for defining 3-Velocity & 4-Velocity in Boyer-Lindquist Coordinates using SI units

Constructor.

Parameters

- **t** (*float*) – Time
- **r** (*float*) – r-Component of 3-Position
- **theta** (*float*) – theta-Component of 3-Position
- **phi** (*float*) – phi-Component of 3-Position
- **v_r** (*float, optional*) – r-Component of 3-Velocity
- **v_th** (*float, optional*) – theta-Component of 3-Velocity

- **v_p** (*float, optional*) – phi-Component of 3-Velocity

position()
Returns Position 4-Vector in SI units

Returns 4-Tuple, containing Position 4-Vector in SI units

Return type tuple

property v_t
Returns the Timelike component of 4-Velocity

velocity(*metric*)
Returns Velocity 4-Vector in SI units

Parameters **metric** (*) – Metric object, in which the coordinates are defined

Returns 4-Tuple, containing Velocity 4-Vector in SI units

Return type tuple

cartesian_differential(**kwargs*)
Converts to Cartesian Coordinates

Parameters ****kwargs** (*dict*) – Keyword Arguments Expects two arguments, `M and ``a`, as described below

Other Parameters

- **M** (*float*) – Mass of the gravitating body, around which, spacetime has been defined
- **a** (*float*) – Spin Parameter of the gravitating body, around which, spacetime has been defined

Returns Cartesian representation of velocity

Return type CartesianDifferential

spherical_differential(**kwargs*)
Converts to Spherical Polar Coordinates

Parameters ****kwargs** (*dict*) – Keyword Arguments Expects two arguments, `M and ``a`, as described below

Other Parameters

- **M** (*float*) – Mass of the gravitating body, around which, spacetime has been defined
- **a** (*float*) – Spin Parameter of the gravitating body, around which, spacetime has been defined

Returns Spherical representation of velocity

Return type SphericalDifferential

1.7.9 Constant module

1.7.10 Units module

einsteinpy.units.**primitive**(**args*)

Strips out units and returns numpy.float64 values out of `astropy.units.quantity.Quantity`

Parameters ***args** (*iterable*) – `astropy.units.quantity.Quantity` objects, who `value` is required

Returns **primitive_args** – List of `numpy.float64` values, obtained from `Quantity` objects

Return type list

1.7.11 Bodies module

Important Bodies. Contains some predefined bodies of the Solar System: * Sun () * Earth () * Moon () * Mercury () * Venus () * Mars () * Jupiter () * Saturn () * Uranus () * Neptune () * Pluto () and a way to define new bodies (*Body* class). Data references can be found in *constant*

class einsteinpy.bodies.**Body**(*name='Generic Body'*, *mass=<Quantity 0. kg>*, *q=<Quantity 0. C>*, *R=<Quantity 0. km>*, *differential=None*, *parent=None*)

Bases: `object`

Class to create a generic Body

Parameters

- **name** (*string*) – Name or ID of the body
- **mass** (*kg*) – Mass of the body
- **q** (*C, optional*) – Charge on the body
- **R** (*units*) – Radius of the body
- **differential** (**, optional*) – Complete coordinates of the body
- **parent** (`Body`, *optional*) – The parent object of the body Useful in case of multibody systems

1.7.12 Geodesic module

This module contains the class, defining a general Geodesic:

class einsteinpy.geodesic.**Geodesic**(*position*, *momentum*, *a=0.0*, *end_lambda=50.0*, *step_size=0.0005*, *time_like=True*, *return_cartesian=True*, *julia=True*)

Bases: `object`

Base Class for defining Geodesics Working in Geometrized Units (M-Units), with `G = c = M = 1.`

Constructor

Parameters

- **position** (*array_like*) – Length-3 Array, containing the initial 3-Position
- **momentum** (*array_like*) – Length-3 Array, containing the initial 3-Momentum
- **a** (*float, optional*) – Dimensionless Spin Parameter of the Black Hole `0 <= a <= 1` Defaults to `0.` (Schwarzschild Black Hole)

- **end_lambda** (*float, optional*) – Affine Parameter value, where integration will end Equivalent to Proper Time for Timelike Geodesics Defaults to `50.`
- **step_size** (*float, optional*) – Size of each geodesic integration step A fixed-step, symplectic VerletLeapfrog integrator is used Defaults to `0.0005`
- **time_like** (*bool, optional*) – Determines type of Geodesic `True` for Time-like geodesics `False` for Null-like geodesics Defaults to `True`
- **return_cartesian** (*bool, optional*) – Whether to return calculated positions in Cartesian Coordinates This only affects the coordinates. The momenta dimensionless quantities, and are returned in Spherical Polar Coordinates. Defaults to `True`
- **julia** (*bool, optional*) – Whether to use the julia backend Defaults to `True`

property trajectory
Returns the trajectory of the test particle

calculate_trajectory()
Calculate trajectory in spacetime, according to Geodesic Equations

Returns

- *~numpy.ndarray* – N-element numpy array, containing affine parameter values, where the integration was performed
- *~numpy.ndarray* – Shape-(N, 6) numpy array, containing [x1, x2, x3, p_r, p_theta, p_phi] for each Lambda

class einsteinpy.geodesic.**Nulllike**(*position*, *momentum*, *a=0.0*, *end_lambda=50.0*, *step_size=0.0005*, *return_cartesian=True*, *julia=True*)
Bases: `einsteinpy.geodesic.geodesic.Geodesic`

Class for defining Null-like Geodesics

Constructor

Parameters

- **position** (*array_like*) – Length-3 Array, containing the initial 3-Position
- **momentum** (*array_like*) – Length-3 Array, containing the initial 3-Momentum
- **a** (*float, optional*) – Dimensionless Spin Parameter of the Black Hole `0 <= a <= 1` Defaults to `0.` (Schwarzschild Black Hole)
- **end_lambda** (*float, optional*) – Affine Parameter value, where integration will end Equivalent to Proper Time for Timelike Geodesics Defaults to `50.`
- **step_size** (*float, optional*) – Size of each geodesic integration step A fixed-step, symplectic VerletLeapfrog integrator is used Defaults to `0.0005`
- **return_cartesian** (*bool, optional*) – Whether to return calculated positions in Cartesian Coordinates This only affects the coordinates. The momenta dimensionless quantities, and are returned in Spherical Polar Coordinates. Defaults to `True`
- **julia** (*bool, optional*) – Whether to use the julia backend Defaults to `True`

class einsteinpy.geodesic.**Timelike**(*position*, *momentum*, *a=0.0*, *end_lambda=50.0*, *step_size=0.0005*, *return_cartesian=True*, *julia=True*)
Bases: `einsteinpy.geodesic.geodesic.Geodesic`

Class for defining Time-like Geodesics

Constructor

Parameters

- **position** (*array_like*) – Length-3 Array, containing the initial 3-Position
- **momentum** (*array_like*) – Length-3 Array, containing the initial 3-Momentum
- **a** (*float, optional*) – Dimensionless Spin Parameter of the Black Hole `0 <= a <= 1` Defaults to `0.` (Schwarzschild Black Hole)
- **end_lambda** (*float, optional*) – Affine Parameter value, where integration will end Equivalent to Proper Time for Timelike Geodesics Defaults to `50.`
- **step_size** (*float, optional*) – Size of each geodesic integration step A fixed-step, symplectic VerletLeapfrog integrator is used Defaults to `0.0005`
- **return_cartesian** (*bool, optional*) – Whether to return calculated positions in Cartesian Coordinates This only affects the coordinates. The momenta dimensionless quantities, and are returned in Spherical Polar Coordinates. Defaults to `True`
- **julia** (*bool, optional*) – Whether to use the julia backend Defaults to `True`

1.7.13 Examples module

This module contains examples, that showcase how the various Numerical Relativity modules in EinsteinPy come together:

`einsteinpy.examples.`**precession**`()`

An example to showcase the usage of the various modules in `einsteinpy`. Here, we assume a Schwarzschild spacetime and obtain a test particle orbit, that shows apsidal precession.

Returns **geod** – Timelike Geodesic, defining test particle trajectory

Return type *Timelike*

1.8 Code of Conduct

The community of participants in einsteinpy is made up of members from around the globe with a diverse set of skills, personalities, and experiences. It is through these differences that our community experiences success and continued growth. We expect everyone in our community to follow these guidelines when interacting with others both inside and outside of our community. Our goal is to keep ours a positive, inclusive, successful, and growing community.

As members of the community,

- We pledge to treat all people with respect and provide a harassment- and bullying-free environment, regardless of sex, sexual orientation and/or gender identity, disability, physical appearance, body size, race, nationality, ethnicity, and religion. In particular, sexual language and imagery, sexist, racist, or otherwise exclusionary jokes are not appropriate.
- We pledge to respect the work of others by recognizing acknowledgment/citation requests of original authors. As authors, we pledge to be explicit about how we want our own work to be cited or acknowledged.
- We pledge to welcome those interested in joining the community, and realize that including people with a variety of opinions and backgrounds will only serve to enrich our community. In particular, discussions relating to pros/cons of various technologies, programming languages, and so on are welcome, but these should be done with respect, taking proactive measure to ensure that all participants are heard and feel confident that they can freely express their opinions.

- We pledge to welcome questions and answer them respectfully, paying particular attention to those new to the community. We pledge to provide respectful criticisms and feedback in forums, especially in discussion threads resulting from code contributions.
- We pledge to be conscientious of the perceptions of the wider community and to respond to criticism respectfully. We will strive to model behaviors that encourage productive debate and disagreement, both within our community and where we are criticized. We will treat those outside our community with the same respect as people within our community.
- We pledge to help the entire community follow the code of conduct, and to not remain silent when we see violations of the code of conduct. We will take action when members of our community violate this code such as contacting shreyas@einsteinpy.org (all emails sent to this address will be treated with the strictest confidence) or talking privately with the person.

This code of conduct applies to all community situations online and offline, including mailing lists, forums, social media, conferences, meetings, associated social events, and one-to-one interactions.

Parts of this code of conduct have been adapted from the Astropy code of conduct.

PYTHON MODULE INDEX

e

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W