EinsteinPy

Release 0.2.1

unknown

Nov 02, 2019

CONTENTS

EinsteinPy is an open source pure Python package dedicated to problems arising in General Relativity and gravitational physics, such as goedesics 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

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 StaticGeodesicPlotter
a = StaticGeodesicPlotter(mass)
a.plot(r,v)
```

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/
$ python setup.py 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 dynamics. These geometry has a central operating quantity known as metric tensor and encapsulate all the geometrical and topological information about the 4d spacetime in them.

- The central quantity required to simulate trajectory of a particle in a gravitational field is christoffel symbols.
- EinsteinPy provides an easy to use interface to calculate these symbols.

Schwarzschild metric

EinsteinPy provides an easy interface for calculating time-like geodesics in Schwarzschild Geometry.

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

```
import numpy as np
from astropy import units as u
from einsteinpy.coordinates import SphericalDifferential,␣
↪CartesianDifferential
from einsteinpy.metric import Schwarzschild
```

From position and velocity in Spherical Coordinates

There are several methods available to create *Schwarzschild* objects. For example, if we have the position and velocity vectors we can use `from_spherical()`:

```
M = 5.972e24 * u.kg
sph_coord = SphericalDifferential(306.0 * u.m, np.pi/2 * u.rad, -np.pi/6*u.
↪rad,
                            0*u.m/u.s, 0*u.rad/u.s, 1900*u.rad/u.s)
obj = Schwarzschild.from_coords(sph_coord, M , 0* u.s)
```

From position and velocity in Cartesian Coordinates

For initializing with Cartesian Coordinates, we can use `from_cartesian`:

```
cartsn_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)
obj = Schwarzschild.from_coords(cartsn_coord, M , 0* u.s)
```

Calculating Trajectory/Time-like Geodesics

After creating the object we can call *calculate_trajectory*

```
end_tau = 0.01 # approximately equal to coordinate time
stepsize = 0.3e-6
ans = obj.calculate_trajectory(end_lambda=end_tau, OdeMethodKwargs={"stepsize
↪":stepsize})
print(ans)
```

```
(array([0.00000000e+00, 2.40000000e-07, 2.64000000e-06, ...,
    9.99367909e-03, 9.99607909e-03, 9.99847909e-03]), array([[ 0.
↪00000000e+00,  3.06000000e+02,  1.57079633e+00, ...,
        0.00000000e+00,  0.00000000e+00,  9.50690000e+02],
    [ 2.39996635e-07,  3.05999885e+02,  1.57079633e+00, ...,
        -9.55164950e+02,  1.32822112e-17,  9.50690712e+02],
    [ 2.63996298e-06,  3.05986131e+02,  1.57079633e+00, ...,
        -1.05071184e+04,  1.46121838e-16,  9.50776184e+02],
    ...,
```

(continues on next page)

(continued from previous page)

```
    [ 9.99381048e-03,  3.05156192e+02,  1.57079633e+00, ...,
        8.30642520e+04, -1.99760372e-12,  9.55955926e+02],
    [ 9.99621044e-03,  3.05344028e+02,  1.57079633e+00, ...,
        7.34673728e+04, -2.01494258e-12,  9.54780155e+02],
    [ 9.99861041e-03,  3.05508844e+02,  1.57079633e+00, ...,
        6.38811856e+04, -2.03252073e-12,  9.53750261e+02]]))
```

Return value can be obtained in Cartesian Coordinates by :

```
ans = obj.calculate_trajectory(end_lambda=end_tau, OdeMethodKwargs={"stepsize
↪":stepsize}, return_cartesian=True)
```

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 ScatterGeodesicPlotter
obj = ScatterGeodesicPlotter()
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)

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:

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

  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 Visualizing advancement of perihelion in Schwarzschild space-time

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

from plotly.offline import init_notebook_mode

from einsteinpy.plotting import GeodesicPlotter
from einsteinpy.coordinates import SphericalDifferential
from einsteinpy.bodies import Body
from einsteinpy.geodesic import Geodesic
```

[2]:
```
init_notebook_mode(connected=True)
# Essential when using Jupyter Notebook (May skip in Jupyter Lab)
```

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

Defining various parameters

- Mass of the attractor(M)
- Initial position and velocity vectors of test partcle

[3]:
```
Attractor = Body(name="BH", mass=6e24 * u.kg, parent=None)
sph_obj = SphericalDifferential(130*u.m, np.pi/2*u.rad, -np.pi/8*u.rad,
                                0*u.m/u.s, 0*u.rad/u.s, 1900*u.rad/u.s)
Object = Body(differential=sph_obj, parent=Attractor)
geodesic = Geodesic(body=Object, time=0 * u.s, end_lambda=0.002, step_size=5e-8)
```

Plotting the trajectory

[4]:
```
obj = GeodesicPlotter()
obj.plot(geodesic)
obj.show()
```

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

It can be seen that the orbit advances along the azimuth angle on each revolution of test partcle .

1.4.2 Animations in EinsteinPy

Import the required modules

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

from einsteinpy.plotting import StaticGeodesicPlotter
from einsteinpy.coordinates import SphericalDifferential
from einsteinpy.bodies import Body
from einsteinpy.geodesic import Geodesic
```

Defining various parameters

- Mass of the attractor (M)
- Initial position and velocity vectors of test partcle

[2]:
```
Attractor = Body(name="BH", mass=6e24 * u.kg, parent=None)
sph_obj = SphericalDifferential(130*u.m, np.pi/2*u.rad, -np.pi/8*u.rad,
                                0*u.m/u.s, 0*u.rad/u.s, 1900*u.rad/u.s)
Object = Body(differential=sph_obj, parent=Attractor)
geodesic = Geodesic(body=Object, time=0 * u.s, end_lambda=0.002, step_size=5e-8)
```

Plotting the animation

[3]:
```
%matplotlib notebook
obj = StaticGeodesicPlotter()
obj.animate(geodesic, interval=25)
obj.show()
```

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

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

[ ]:

1.4.3 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]:
$$\left[\begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{bmatrix}\right.$$

(continues on next page)

(continued from previous page)

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

[6]:
$$\begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{bmatrix}$$

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

[8]:
```
# single substitution
subs1 = sch.subs(a,0)
subs1.tensor()
```

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

[9]:
```
# multiple substitution
subs2 = sch.subs([(a,0), (c,1)])
subs2.tensor()
```

[9]:
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -r^2 & 0 \\ 0 & 0 & 0 & -r^2 \sin^2(\theta) \end{bmatrix}$$

(continues on next page)

(continued from previous page)

[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]:

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

1.4.4 Contravariant & Covariant indices in Tensors (Symbolic)

[1]:
```
from einsteinpy.symbolic import SchwarzschildMetric, MetricTensor, ChristoffelSymbols,
↪ RiemannCurvatureTensor
import sympy
sympy.init_printing()
```

Analysing the schwarzschild metric along with performing various operations

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

[2]:

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

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

[3]:

$$\begin{bmatrix} \frac{r}{-a+r} & 0 & 0 & 0 \\ 0 & \frac{c^2(a-r)}{r} & 0 & 0 \\ 0 & 0 & -\frac{c^2}{r^2} & 0 \\ 0 & 0 & 0 & -\frac{c^2}{r^2\sin^2(\theta)} \end{bmatrix}$$

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

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

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

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

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

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

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

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

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

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

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

[14]:
$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \frac{ac^2}{r^3} & 0 & 0 \\ 0 & 0 & -\frac{ac^2}{2r^3} & 0 \\ 0 & 0 & 0 & -\frac{ac^2}{2r^3} \end{bmatrix}$$

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

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

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

1.4.5 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

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 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.expr
```

[4]:

$$-12$$

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

1.4.6 Analysing Earth using EinsteinPy!

Calculating the eccentricity and speed at apehelion of Earth's orbit

Various parameters of Earth's orbit considering Sun as schwarzschild body and solving geodesic equations are calculated

1. Defining the initial parameters

[1]:
```
from astropy import units as u
import numpy as np
from einsteinpy.metric import Schwarzschild
from einsteinpy.coordinates import SphericalDifferential
```

[2]:
```
# Define position and velocity vectors in spherical coordinates
# Earth is at perihelion
M = 1.989e30 * u.kg  # mass of sun
distance = 147.09e6 * u.km
speed_at_perihelion = 30.29 * u.km / u.s
omega = (u.rad * speed_at_perihelion) / distance

sph_obj = SphericalDifferential(distance, np.pi / 2 * u.rad, np.pi * u.rad,
                                0 * u.km / u.s, 0 * u.rad / u.s, omega)
```

- Defining λ (or τ) for which to calculate trajectory
  - λ is proper time and is approximately equal to coordinate time in non-relativistic limits

[3]:
```
# Set lambda to complete an year.
# Lambda is always specified in secs
end_lambda = ((1 * u.year).to(u.s)).value
# Choosing stepsize for ODE solver to be 5 minutes
stepsize = ((5 * u.min).to(u.s)).value
```

2. Making a class instance to get the trajectory in cartesian coordinates

[4]:
```
obj = Schwarzschild.from_coords(sph_obj, M)
ans = obj.calculate_trajectory(
    end_lambda=end_lambda, OdeMethodKwargs={"stepsize": stepsize}, return_
↪cartesian=True
)
```

- Return value is a tuple consisting of 2 numpy array
  - First one contains list of λ
  - Seconds is array of shape (n,8) where each component is:
  - t - coordinate time
  - x - position in m
  - y - position in m
  - z - position in m
  - dt/dλ
  - dx/dλ
  - dy/dλ
  - dz/dλ

[5]:
```
ans[0].shape, ans[1].shape
```

[5]:
```
((13150,), (13150, 8))
```

Calculating distance at aphelion

- Should be 152.10 million km

[6]:
```
r = np.sqrt(np.square(ans[1][:, 1]) + np.square(ans[1][:, 2]))
i = np.argmax(r)
(r[i] * u.m).to(u.km)
```

[6]:
1.5205967×10^{8} km

Speed at aphelion should be 29.29 km/s and should be along y-axis

[7]:
```
((ans[1][i][6]) * u.m / u.s).to(u.km / u.s)
```

[7]: $29.300051 \ \frac{\mathrm{km}}{\mathrm{s}}$

Calculating the eccentricity

- Should be 0.0167

[8]:
```
xlist, ylist = ans[1][:, 1], ans[1][:, 2]
i = np.argmax(ylist)
x, y = xlist[i], ylist[i]
eccentricity = x / (np.sqrt(x ** 2 + y ** 2))
eccentricity
```

[8]:
```
0.01661834158657882
```

Plotting the trajectory with time

[9]:
```
from einsteinpy.bodies import Body
from einsteinpy.geodesic import Geodesic
Sun = Body(name="Sun", mass=M, parent=None)
Object = Body(name="Earth", differential=sph_obj, parent=Sun)
geodesic = Geodesic(body=Object, time=0 * u.s, end_lambda=end_lambda, step_
↪size=stepsize)
from einsteinpy.plotting import GeodesicPlotter
sgp = GeodesicPlotter()
sgp.plot(geodesic)
sgp.show()
```

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

All data regarding earth's orbit is taken from https://nssdc.gsfc.nasa.gov/planetary/factsheet/earthfact.html

1.4.7 Visualizing frame dragging in Kerr space-time

Importing required modules

[1]:
```
from astropy import units as u
import numpy as np
from einsteinpy.metric import Kerr
from einsteinpy.coordinates import BoyerLindquistDifferential
```

Defining position/velocity of test particle

- Initial velocity is kept 0

[2]:
```
M = 1.989e30 * u.kg
a = 0.3 * u.m
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,
                                    a)
```

[3]:
```
end_lambda = ((1 * u.year).to(u.s)).value / 930
# Choosing stepsize for ODE solver to be 0.02 minutes
stepsize = ((0.02 * u.min).to(u.s)).value
```

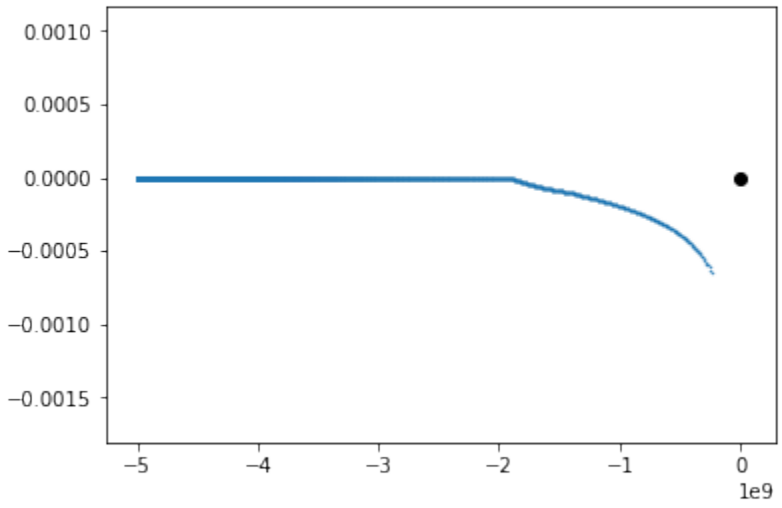
[4]:
```
obj = Kerr.from_coords(BL_obj, M)
ans = obj.calculate_trajectory(
    end_lambda=end_lambda, OdeMethodKwargs={"stepsize": stepsize}, return_
↪cartesian=True
)
x, y = ans[1][:,1], ans[1][:,2]
```

```
/home/shreyas/Softwares/anaconda3/lib/python3.7/site-packages/scipy/integrate/_ivp/
↪common.py:32: UserWarning: The following arguments have no effect for a chosen␣
↪solver: `first_step`.
  .format(", ".join("`{}`".format(x) for x in extraneous)))
```

Plotting the trajectory

[5]:
```
%matplotlib inline
```

[6]:
```
import matplotlib.pyplot as plt
plt.scatter(x,y, s=0.2)
plt.scatter(0,0, c='black')
plt.show()
```



It can be seen that as the particle approaches the massive body, it acquires axial velocity due to spin and frame-dragging effect of the body.

1.4.8 Visualizing event horizon and ergosphere of Kerr black hole

Importing required modules

[1]:
```
import numpy as np
import astropy.units as u
import matplotlib.pyplot as plt
from einsteinpy.utils import kerr_utils, schwarzschild_radius
```

Defining the black hole charecteristics

[2]:
```
M = 4e30
scr = schwarzschild_radius(M * u.kg).value
# for nearly maximally rotating black hole
a1 = 0.499999*scr
# for ordinary black hole
a2 = 0.3*scr
```

Calculating the ergosphere and event horizon for spherical coordinates

[3]:
```
ergo1, ergo2, hori1, hori2 = list(), list(), list(), list()
thetas = np.linspace(0, np.pi, 720)
for t in thetas:
    ergo1.append(kerr_utils.radius_ergosphere(M, a1, t, "Spherical"))
    ergo2.append(kerr_utils.radius_ergosphere(M, a2, t, "Spherical"))
    hori1.append(kerr_utils.event_horizon(M, a1, t, "Spherical"))
    hori2.append(kerr_utils.event_horizon(M, a2, t, "Spherical"))
ergo1, ergo2, hori1, hori2 = np.array(ergo1), np.array(ergo2), np.array(hori1), np.
↪array(hori2)
```

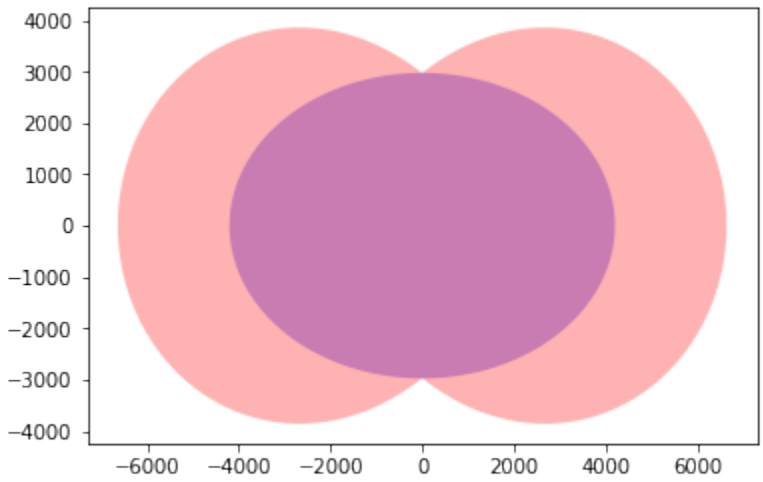
Calculating the X, Y coordinates for plotting

[4]:
```
Xe1, Ye1 = ergo1[:,0] * np.sin(ergo1[:,1]), ergo1[:,0] * np.cos(ergo1[:,1])
Xh1, Yh1 = hori1[:,0] * np.sin(hori1[:,1]), hori1[:,0] * np.cos(hori1[:,1])
Xe2, Ye2 = ergo2[:,0] * np.sin(ergo2[:,1]), ergo2[:,0] * np.cos(ergo2[:,1])
Xh2, Yh2 = hori2[:,0] * np.sin(hori2[:,1]), hori2[:,0] * np.cos(hori2[:,1])
```

Plot for maximally rotating black hole

[5]:
```
%matplotlib inline
fig, ax = plt.subplots()
# for maximally rotating black hole
ax.fill(Xh1, Yh1, 'b', Xe1, Ye1, 'r', alpha=0.3)
ax.fill(-1*Xh1, Yh1, 'b', -1*Xe1, Ye1, 'r', alpha=0.3)
```

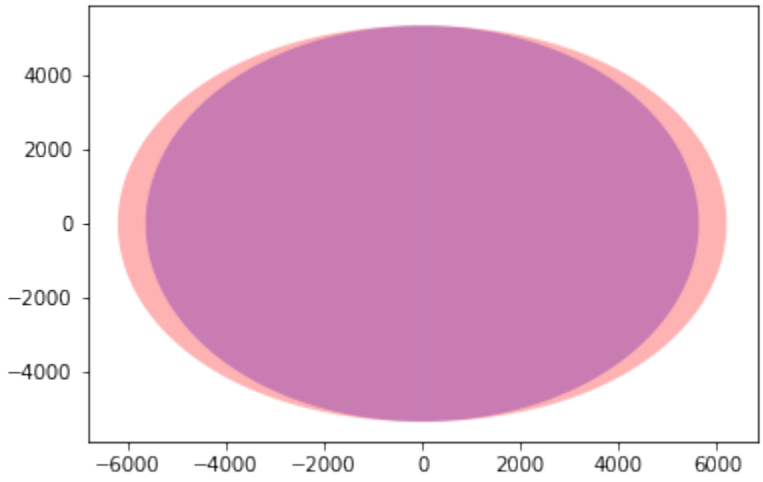
[5]:
```
[<matplotlib.patches.Polygon at 0x7fa4665277b8>,
 <matplotlib.patches.Polygon at 0x7fa466527dd8>]
```



Plot for rotating(normally) black hole

[6]:
```
%matplotlib inline
fig, ax = plt.subplots()
ax.fill(Xh2, Yh2, 'b', Xe2, Ye2, 'r', alpha=0.3)
ax.fill(-1*Xh2, Yh2, 'b', -1*Xe2, Ye2, 'r', alpha=0.3)
```

[6]:
```
[<matplotlib.patches.Polygon at 0x7fa466598160>,
 <matplotlib.patches.Polygon at 0x7fa438289ba8>]
```



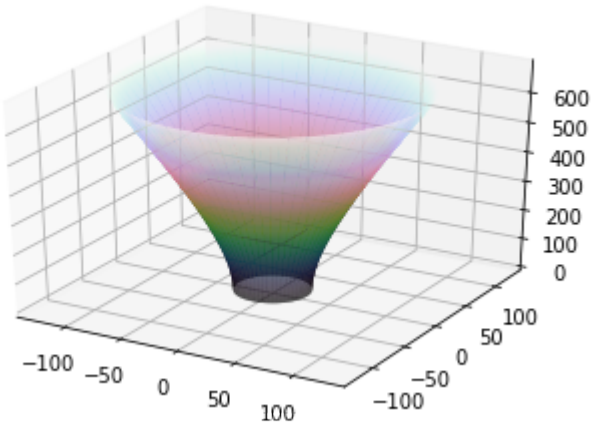
- The inner body represents event horizon and outer one represents ergosphere. It can be concluded that with decrease in angular momentum, radius of event horizon increases, and that of ergosphere decreases.

1.4.9 Plotting Spacial Hypersurface Embedding for Schwarzschild Space-Time

[1]:
```
from einsteinpy.hypersurface import SchwarzschildEmbedding
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)
surface_obj.plot_hypersurface(plot_type='surface')
surface_obj.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)!

1.4.10 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]:

$$\left[\begin{bmatrix}0 & 0 & 0 & 0\\ 0 & -\cos^2(t) & 0 & 0\\ 0 & 0 & -\cos^2(t)\sinh^2(\chi) & 0\\ 0 & 0 & 0 & -\sin^2(\theta)\cos^2(t)\sinh^2(\chi)\end{bmatrix}, \begin{bmatrix}0 & 0 & 0 & 0\\ \cos^2(t) & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{bmatrix}, \begin{bmatrix}0 & \cos^2(t) & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{bmatrix}, \begin{bmatrix}-\cos^2(t) & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & \cos^4(t)\sinh^2(\chi) & 0\\ 0 & 0 & 0 & \sin^2(\theta)\cos^4(t)\sinh^2(\chi)\end{bmatrix},$$

$$\begin{bmatrix}0 & 0 & \cos^2(t)\sinh^2(\chi) & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{bmatrix}, \begin{bmatrix}0 & 0 & 0 & 0\\ 0 & 0 & -\cos^4(t)\sinh^2(\chi) & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{bmatrix},$$

$$\begin{bmatrix}0 & 0 & 0 & \sin^2(\theta)\cos^2(t)\sinh^2(\chi)\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{bmatrix}, \begin{bmatrix}0 & 0 & 0 & 0\\ 0 & 0 & 0 & -\sin^2(\theta)\cos^4(t)\sinh^2(\chi)\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{bmatrix}\right]$$

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

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

1.4.11 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]:

$$\begin{bmatrix}-3.0 & 0 & 0 & 0\\ 0 & 3.0\cos^2(t) & 0 & 0\\ 0 & 0 & \left(\sin^2(t)-1\right)\sinh^2(\chi)+4.0\cos^2(t)\sinh^2(\chi) & 0\\ 0 & 0 & 0 & \left(\sin^2(t)-1\right)\sin^2(\theta)\sinh^2(\chi)+4.0\sin^2(\theta)\cos^2(t)\sinh^2(\chi)\end{bmatrix}$$

1.4.12 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]:
$$(x, \quad y)$$

`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]:
$$[[2, \quad 1], \quad [0, \quad 3]]$$

1.5 What's new

1.5.1 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 porjects 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+

1.5.2 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+

1.5.3 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`.

1.5.4 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.

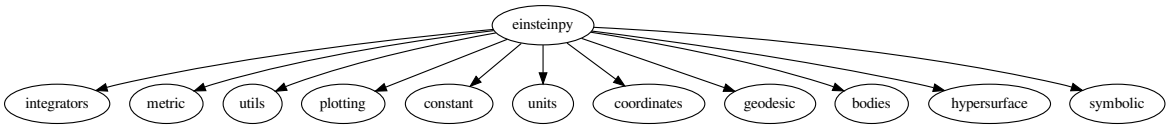
But to avoid confusion, we have setup tox for doing this in one command and doing it properly! Run:

```
$ 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*)

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*)

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.

schwarzschild module

This module contains the basic class for calculating time-like geodesics in Schwarzschild Space-Time:

class einsteinpy.metric.schwarzschild.**Schwarzschild**(*sph_coords*, *M*, *time*)
Class for defining a Schwarzschild Geometry methods

classmethod from_coords(*coords*, *M*, *q=None*, *Q=None*, *time=<Quantity 0. s>*, *a=<Quantity 0. m>*)
Constructor

Parameters

- **coords** (CartesianDifferential) – Object having both initial positions and velocities of particle in Cartesian Coordinates
- **M** (*Quantity*) – Mass of the body
- **time** (*Quantity*) – Time of start, defaults to 0 seconds.

calculate_trajectory(*start_lambda=0.0*, *end_lambda=10.0*, *stop_on_singularity=True*, *OdeMethodKwargs={'stepsize': 0.001}*, *return_cartesian=False*)
Calculate trajectory in manifold according to geodesic equation

Parameters

- **start_lambda** (*float*) – Starting lambda(proper time), defaults to 0, (lambda ~= t)
- **end_lamdba** (*float*) – Lambda(proper time) where iteartions will stop, defaults to 100000
- **stop_on_singularity** (*bool*) – Whether to stop further computation on reaching singularity, defaults to True
- **OdeMethodKwargs** (*dict*) – Kwargs to be supplied to the ODESolver, defaults to {'stepsize': 1e-3} Dictionary with key 'stepsize' along with an float value is expected.
- **return_cartesian** (*bool*) – True if coordinates and velocities are required in cartesian coordinates(SI units), defaults to False

Returns

- *~numpy.ndarray* – N-element array containing proper time.
- *~numpy.ndarray* – (n,8) shape array of [t, x1, x2, x3, velocity_of_time, v1, v2, v3] for each proper time(lambda).

calculate_trajectory_iterator(*start_lambda=0.0*, *stop_on_singularity=True*, *OdeMethodKwargs={'stepsize': 0.001}*, *return_cartesian=False*)

Calculate trajectory in manifold according to geodesic equation Yields an iterator

Parameters

- **start_lambda** (*float*) – Starting lambda, defaults to 0.0, (lambda ~= t)
- **stop_on_singularity** (*bool*) – Whether to stop further computation on reaching singularity, defaults to True
- **OdeMethodKwargs** (*dict*) – Kwargs to be supplied to the ODESolver, defaults to {'stepsize': 1e-3} Dictionary with key 'stepsize' along with an float value is expected.
- **return_cartesian** (*bool*) – True if coordinates and velocities are required in cartesian coordinates(SI units), defaults to Falsed

Yields

- *float* – proper time
- *~numpy.ndarray* – array of [t, x1, x2, x3, velocity_of_time, v1, v2, v3] for each proper time(lambda).

kerr module

This module contains the basic class for calculating time-like geodesics in Kerr Space-Time:

class einsteinpy.metric.kerr.**Kerr**(*bl_coords*, *M*, *time*)

Class for defining Kerr Goemetry Methdos

classmethod from_coords(*coords*, *M*, *q=None*, *Q=None*, *time=<Quantity 0. s>*, *a=<Quantity 0. m>*)

Constructor

Parameters

- **coords** (CartesianDifferential) – Object having both initial positions and velocities of particle in Cartesian Coordinates
- **M** (*Quantity*) – Mass of the body
- **a** (*Quantity*) – Spin factor of the massive body. Angular momentum divided by mass divided by speed of light.
- **time** (*Quantity*) – Time of start, defaults to 0 seconds.

calculate_trajectory(*start_lambda=0.0*, *end_lambda=10.0*, *stop_on_singularity=True*, *OdeMethodKwargs={'stepsize': 0.001}*, *return_cartesian=False*)

Calculate trajectory in manifold according to geodesic equation

Parameters

- **start_lambda** (*float*) – Starting lambda(proper time), defaults to 0, (lambda ~= t)
- **end_lamdba** (*float*) – Lambda(proper time) where iteartions will stop, defaults to 100000
- **stop_on_singularity** (*bool*) – Whether to stop further computation on reaching singularity, defaults to True

- **OdeMethodKwargs** (*dict*) – Kwargs to be supplied to the ODESolver, defaults to {'stepsize': 1e-3} Dictionary with key 'stepsize' along with an float value is expected.
- **return_cartesian** (*bool*) – True if coordinates and velocities are required in cartesian coordinates(SI units), defaults to False

Returns

- *~numpy.ndarray* – N-element array containing proper time.
- *~numpy.ndarray* – (n,8) shape array of [t, x1, x2, x3, velocity_of_time, v1, v2, v3] for each proper time(lambda).

calculate_trajectory_iterator(*start_lambda=0.0*, *stop_on_singularity=True*, *OdeMethodKwargs={'stepsize': 0.001}*, *return_cartesian=False*)

Calculate trajectory in manifold according to geodesic equation. Yields an iterator.

Parameters

- **start_lambda** (*float*) – Starting lambda, defaults to 0.0, (lambda ~= t)
- **stop_on_singularity** (*bool*) – Whether to stop further computation on reaching singularity, defaults to True
- **OdeMethodKwargs** (*dict*) – Kwargs to be supplied to the ODESolver, defaults to {'stepsize': 1e-3} Dictionary with key 'stepsize' along with an float value is expected.
- **return_cartesian** (*bool*) – True if coordinates and velocities are required in cartesian coordinates(SI units), defaults to Falsed

Yields

- *float* – proper time
- *~numpy.ndarray* – array of [t, x1, x2, x3, velocity_of_time, v1, v2, v3] for each proper time(lambda).

kerr-newman module

This module contains the basic class for calculating time-like geodesics in Kerr-Newman Space-Time:

class einsteinpy.metric.kerrnewman.**KerrNewman**(*bl_coords*, *q*, *M*, *Q*, *time*)

Class for defining Kerr-Newman Goemetry Methods

classmethod from_coords(*coords*, *M*, *q*, *Q*, *time=<Quantity 0. s>*, *a=<Quantity 0. m>*)

Constructor.

Parameters

- **coords** (BoyerLindquistDifferential) – Initial positions and velocities of particle in BL Coordinates, and spin factor of massive body.
- **q** (*Quantity*) – Charge per unit mass of test particle
- **M** (*Quantity*) – Mass of the massive body
- **Q** (*Quantity*) – Charge on the massive body
- **a** (*Quantity*) – Spin factor of the massive body(Angular Momentum per unit mass per speed of light)
- **time** (*Quantity*) – Time of start, defaults to 0 seconds.

calculate_trajectory(*start_lambda=0.0*, *end_lambda=10.0*, *stop_on_singularity=True*, *OdeMethodKwargs={'stepsize': 0.001}*, *return_cartesian=False*)

Calculate trajectory in manifold according to geodesic equation

Parameters

- **start_lambda** (*float*) – Starting lambda(proper time), defaults to 0.0, (lambda ~= t)
- **end_lamdba** (*float*) – Lambda(proper time) where iteartions will stop, defaults to 100000
- **stop_on_singularity** (*bool*) – Whether to stop further computation on reaching singularity, defaults to True
- **OdeMethodKwargs** (*dict*) – Kwargs to be supplied to the ODESolver, defaults to {'stepsize': 1e-3} Dictionary with key 'stepsize' along with an float value is expected.
- **return_cartesian** (*bool*) – True if coordinates and velocities are required in cartesian coordinates(SI units), defaults to False

Returns

- *~numpy.ndarray* – N-element array containing proper time.
- *~numpy.ndarray* – (n,8) shape array of [t, x1, x2, x3, velocity_of_time, v1, v2, v3] for each proper time(lambda).

calculate_trajectory_iterator(*start_lambda=0.0*, *stop_on_singularity=True*, *OdeMethodKwargs={'stepsize': 0.001}*, *return_cartesian=False*)

Calculate trajectory in manifold according to geodesic equation. Yields an iterator.

Parameters

- **start_lambda** (*float*) – Starting lambda, defaults to 0.0, (lambda ~= t)
- **stop_on_singularity** (*bool*) – Whether to stop further computation on reaching singularity, defaults to True
- **OdeMethodKwargs** (*dict*) – Kwargs to be supplied to the ODESolver, defaults to {'stepsize': 1e-3} Dictionary with key 'stepsize' along with an float value is expected.
- **return_cartesian** (*bool*) – True if coordinates and velocities are required in cartesian coordinates(SI units), defaults to Falsed

Yields

- *float* – proper time
- *~numpy.ndarray* – array of [t, x1, x2, x3, velocity_of_time, v1, v2, v3] for each proper time(lambda).

1.7.3 Symbolic Relativity Module

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

Predefined Metrics

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

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

Symbolic Constants Module

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

class einsteinpy.symbolic.constants.**SymbolicConstant**
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'.

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:

class einsteinpy.symbolic.tensor.**Tensor**(*arr*, *config='ll'*)
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'.

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'

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()
Returns a simplified Tensor

Returns Simplified Tensor

Return type *Tensor*

class einsteinpy.symbolic.tensor.**BaseRelativityTensor**(*arr*, *syms*, *config='ll'*, *parent_metric=None*, *variables=[]*, *functions=[]*, *name=None*)
Inherits from ~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.

Type string 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** (*string*) – Name of the Tensor. Defaults to None.

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.

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.

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='l'*, *parent_metric=None*)
Inherits from ~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 'l'.
- **parent_metric** (*MetricTensor or None*) – Corresponding Metric for the Generic Vector. Defaults to None.

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.

Metric Tensor Module

This module contains the class for defining a Metric belonging to any arbitrary space-time symbolically:

class einsteinpy.symbolic.metric.**MetricTensor**(*arr*, *syms*, *config='ll'*)

Inherits from ~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'.

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*

Vacuum Solutions Module

This module contains various exact vacuum solutions to Einstein's Field Equation in form of metric tensor:

einsteinpy.symbolic.vacuum_metrics.**SchwarzschildMetric**(*symbolstr='t r theta phi'*)

Returns Metric Tensor of symbols of Schwarzschild Metric.

Parameters **symbolstr** (*string*) – symbols to be used to define schwarzschild space, defaults to 't r theta phi'

Returns Metric Tensor for Schwarzschild space-time

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*)

Inherits from ~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.

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.

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*)

Inherits from ~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.

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.

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*)

Inherits from ~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.

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.

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

Inherits from ~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*)

Inherits from ~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.

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.

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*)

Inherits from ~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.

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.

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*)

Inherits from ~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.

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.

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*)
Inherits from ~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.

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.

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*)
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

plot_hypersurface(*plot_type='wireframe'*, *alpha=100*)
Plots the surface thus obtained for the embedding.

Parameters

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

show()
Show the plot made by plot_hypersurface()

1.7.5 Utils module

The common utilities of the project are included in this module. This includes easy methods for calculation of the schwartzschild radius, calculation of the rate of expansion of the universe, velocity of time etc.

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'.:

Schwarzschild Geometry Utilities

einsteinpy.utils.schwarzschild_utils.**schwarzschild_radius**(*mass*, *c=<<class 'astropy.constants.codata2014.CODATA2014'> name='Speed of light in vacuum' value=299792458.0 uncertainty=0.0 unit='m / s' reference='CODATA 2014'>*, *G=<<class 'astropy.constants.codata2014.CODATA2014'> name='Gravitational constant' value=6.67408e-11 uncertainty=3.1e-15 unit='m3 / (kg s2)' reference='CODATA 2014'>*)

Schwarzschild radius is the radius defining the event horizon of a Schwarzschild black hole. It is characteristic radius associated with every quantity of mass.

Parameters **mass** (*kg*) –

Returns **r** – Schwarzschild radius for a given mass

Return type m

einsteinpy.utils.schwarzschild_utils.**schwarzschild_radius_dimensionless**(*M*, *c=299792458.0*, *G=6.67408e-11*)

Parameters

- **M** (*float*) – Mass of massive body

- **c** (*float*) – Speed of light, defaults to value of speed of light in SI units.
- **G** (*float*) – Gravitational Constant, defaults to its value in SI units

Returns **Rs** – Schwarzschild radius for a given mass

Return type float

einsteinpy.utils.schwarzschild_utils.**time_velocity**(*pos_vec*, *vel_vec*, *mass*)

Velocity of time calculated from einstein's equation. See http://www.physics.usu.edu/Wheeler/GenRel/Lectures/GRNotesDecSchwarzschildGeodesicsPost.pdf

Parameters

- **pos_vector** (*array*) – Vector with r, theta, phi components in SI units
- **vel_vector** (*array*) – Vector with velocities of r, theta, phi components in SI units
- **mass** (*kg*) – Mass of the body

Returns Velocity of time

Return type one

einsteinpy.utils.schwarzschild_utils.**metric**(*r*, *theta*, *M*, *c=299792458.0*, *G=6.67408e-11*)

Returns the Schwarzschild Metric

Parameters

- **r** (*float*) – Distance from the centre
- **theta** (*float*) – Angle from z-axis
- **M** (*float*) – Mass of the massive body
- **c** (*float*) – Speed of light

Returns Numpy array of shape (4,4)

Return type array

einsteinpy.utils.schwarzschild_utils.**christoffels**(*r*, *theta*, *M*, *c=299792458.0*, *G=6.67408e-11*)

Returns the 3rd rank Tensor containing Christoffel Symbols for Schwarzschild Metric

Parameters

- **r** (*float*) – Distance from the centre
- **theta** (*float*) – Angle from z-axis
- **M** (*float*) – Mass of the massive body
- **c** (*float*) – Speed of light

Returns Numpy array of shape (4,4,4)

Return type array

Kerr Geometry Utilities

einsteinpy.utils.kerr_utils.**nonzero_christoffels_list = [(0, 0, 1), (0, 0, 2), (0, 1, 3),**

Precomputed list of tuples consisting of indices of christoffel symbols which are non-zero in Kerr Metric

einsteinpy.utils.kerr_utils.**scaled_spin_factor**(*a*, *M*, *c=299792458.0*, *G=6.67408e-11*)

Returns a scaled version of spin factor(a)

Parameters

- **a** (*float*) – Number between 0 & 1
- **M** (*float*) – Mass of massive body
- **c** (*float*) – Speed of light. Defaults to speed in SI units.
- **G** (*float*) – Gravitational constant. Defaults to Gravitaional Constant in SI units.

Returns Scaled spinf factor to consider changed units

Return type float

Raises **ValueError** – If a not between 0 & 1

einsteinpy.utils.kerr_utils.**sigma**(*r*, *theta*, *a*)
Returns the value r^2 + a^2 * cos^2(theta) Specific to Boyer-Lindquist coordinates

Parameters

- **r** (*float*) – Component r in vector
- **theta** (*float*) – Component theta in vector
- **a** (*float*) – Any constant

Returns The value r^2 + a^2 * cos^2(theta)

Return type float

einsteinpy.utils.kerr_utils.**delta**(*r*, *M*, *a*, *c=299792458.0*, *G=6.67408e-11*)
Returns the value r^2 - Rs * r + a^2 Specific to Boyer-Lindquist coordinates

Parameters

- **r** (*float*) – Component r in vector
- **M** (*float*) – Mass of massive body
- **a** (*float*) – Any constant

Returns The value r^2 - Rs * r + a^2

Return type float

einsteinpy.utils.kerr_utils.**metric**(*r*, *theta*, *M*, *a*, *c=299792458.0*, *G=6.67408e-11*)
Returns the Kerr Metric

Parameters

- **r** (*float*) – Distance from the centre
- **theta** (*float*) – Angle from z-axis
- **M** (*float*) – Mass of massive body
- **a** (*float*) – Black Hole spin factor
- **c** (*float*) – Speed of light

Returns Numpy array of shape (4,4)

Return type array

einsteinpy.utils.kerr_utils.**metric_inv**(*r*, *theta*, *M*, *a*, *c=299792458.0*, *G=6.67408e-11*)
Returns the inverse of Kerr Metric

Parameters

- **r** (*float*) – Distance from the centre
- **theta** (*float*) – Angle from z-axis
- **M** (*float*) – Mass of massive body
- **a** (*float*) – Black Hole spin factor
- **c** (*float*) – Speed of light

Returns Numpy array of shape (4,4)

Return type array

einsteinpy.utils.kerr_utils.**dmetric_dx**(*r*, *theta*, *M*, *a*, *c=299792458.0*, *G=6.67408e-11*)
Returns differentiation of each component of Kerr metric tensor w.r.t. t, r, theta, phi

Parameters

- **r** (*float*) – Distance from the centre
- **theta** (*float*) – Angle from z-axis
- **M** (*float*) – Mass of massive body
- **a** (*float*) – Black Hole spin factor
- **c** (*float*) – Speed of light

Returns **dmdx** – Numpy array of shape (4,4,4) dmdx[0], dmdx[1], dmdx[2] & dmdx[3] is differentiation of metric w.r.t. t, r, theta & phi respectively

Return type array

einsteinpy.utils.kerr_utils.**christoffels**(*r*, *theta*, *M*, *a*, *c=299792458.0*, *G=6.67408e-11*)
Returns the 3rd rank Tensor containing Christoffel Symbols for Kerr Metric

Parameters

- **r** (*float*) – Distance from the centre
- **theta** (*float*) – Angle from z-axis
- **M** (*float*) – Mass of massive body
- **a** (*float*) – Black Hole spin factor
- **c** (*float*) – Speed of light

Returns Numpy array of shape (4,4,4)

Return type array

einsteinpy.utils.kerr_utils.**kerr_time_velocity**(*pos_vec*, *vel_vec*, *mass*, *a*)
Velocity of coordinate time wrt proper metric

Parameters

- **pos_vector** (*array*) – Vector with r, theta, phi components in SI units
- **vel_vector** (*array*) – Vector with velocities of r, theta, phi components in SI units
- **mass** (*kg*) – Mass of the body
- **a** (*float*) – Any constant

Returns Velocity of time

Return type one

einsteinpy.utils.kerr_utils.**nonzero_christoffels**()
Returns a list of tuples consisting of indices of christoffel symbols which are non-zero in Kerr Metric computed in real-time.

Returns List of tuples each tuple (i,j,k) represent christoffel symbol with i as upper index and j,k as lower indices.

Return type list

einsteinpy.utils.kerr_utils.**spin_factor**(*J*, *M*, *c*)
Calculate spin factor(a) of kerr body

Parameters

- **J** (*float*) – Angular momentum in SI units(kg m2 s-2)
- **M** (*float*) – Mass of body in SI units(kg)
- **c** (*float*) – Speed of light

Returns Spin factor (J/(Mc))

Return type float

einsteinpy.utils.kerr_utils.**event_horizon**(*M*, *a*, *theta=1.5707963267948966*, *coord='BL'*, *c=299792458.0*, *G=6.67408e-11*)
Calculate the radius of event horizon of Kerr black hole

Parameters

- **M** (*float*) – Mass of massive body
- **a** (*float*) – Black hole spin factor
- **theta** (*float*) – Angle from z-axis in Boyer-Lindquist coordinates in radians. Mandatory for coord=='Spherical'. Defaults to pi/2.
- **coord** (*str*) – Output coordinate system. 'BL' for Boyer-Lindquist & 'Spherical' for spherical. Defaults to 'BL'.

Returns [Radius of event horizon(R), angle from z axis(theta)] in BL/Spherical coordinates

Return type array

einsteinpy.utils.kerr_utils.**radius_ergosphere**(*M*, *a*, *theta=1.5707963267948966*, *coord='BL'*, *c=299792458.0*, *G=6.67408e-11*)
Calculate the radius of ergospere of Kerr black hole at a specific azimuthal angle

Parameters

- **M** (*float*) – Mass of massive body
- **a** (*float*) – Black hole spin factor
- **theta** (*float*) – Angle from z-axis in Boyer-Lindquist coordinates in radians. Defaults to pi/2.
- **coord** (*str*) – Output coordinate system. 'BL' for Boyer-Lindquist & 'Spherical' for spherical. Defaults to 'BL'.

Returns [Radius of ergosphere(R), angle from z axis(theta)] in BL/Spherical coordinates

Return type array

Kerr-Newman Geometry Utilities

einsteinpy.utils.kerrnewman_utils.**nonzero_christoffels_list = [(0, 0, 1), (0, 0, 2), (0, 1,**
Precomputed list of tuples consisting of indices of christoffel symbols which are non-zero in Kerr-Newman Metric

einsteinpy.utils.kerrnewman_utils.**charge_length_scale**(*Q*, *c=299792458.0*, *G=6.67408e-11*, *Cc=8987551787.997911*)
Returns a length scale corrosponding to the Electric Charge Q of the mass

Parameters

- **Q** (*float*) – Charge on the massive body
- **c** (*float*) – Speed of light. Defaults to 299792458 (SI units)
- **G** (*float*) – Gravitational constant. Defaults to 6.67408e-11 (SI units)
- **Cc** (*float*) – Coulumb's constant. Defaults to 8.98755e9 (SI units)

Returns returns (coulomb's constant^0.5)*(Q/c^2)*G^0.5

Return type float

einsteinpy.utils.kerrnewman_utils.**rho**(*r*, *theta*, *a*)
Returns the value sqrt(r^2 + a^2 * cos^2(theta)). Specific to Boyer-Lindquist coordinates

Parameters

- **r** (*float*) – Component r in vector
- **theta** (*float*) – Component theta in vector
- **a** (*float*) – Any constant

Returns The value sqrt(r^2 + a^2 * cos^2(theta))

Return type float

einsteinpy.utils.kerrnewman_utils.**delta**(*r*, *M*, *a*, *Q*, *c=299792458.0*, *G=6.67408e-11*, *Cc=8987551787.997911*)
Returns the value r^2 - Rs * r + a^2 Specific to Boyer-Lindquist coordinates

Parameters

- **r** (*float*) – Component r in vector
- **M** (*float*) – Mass of the massive body
- **a** (*float*) – Any constant
- **Q** (*float*) – Charge on the massive body
- **c** (*float*) – Speed of light
- **G** (*float*) – Gravitational constant
- **Cc** (*float*) – Coulomb's constant

Returns The value r^2 - Rs * r + a^2 + Rq^2

Return type float

einsteinpy.utils.kerrnewman_utils.**metric**(*r*, *theta*, *M*, *a*, *Q*, *c=299792458.0*, *G=6.67408e-11*, *Cc=8987551787.997911*)
Returns the Kerr-Newman Metric

Parameters

- **r** (*float*) – Distance from the centre
- **theta** (*float*) – Angle from z-axis
- **M** (*float*) – Mass of the massive body
- **a** (*float*) – Black Hole spin factor
- **Q** (*float*) – Charge on the massive body
- **c** (*float*) – Speed of light
- **G** (*float*) – Gravitational constant
- **Cc** (*float*) – Coulomb's constant

Returns Numpy array of shape (4,4)

Return type array

einsteinpy.utils.kerrnewman_utils.**metric_inv**(*r*, *theta*, *M*, *a*, *Q*, *c=299792458.0*, *G=6.67408e-11*, *Cc=8987551787.997911*)

Returns the inverse of Kerr-Newman Metric

Parameters

- **r** (*float*) – Distance from the centre
- **theta** (*float*) – Angle from z-axis
- **M** (*float*) – Mass of the massive body
- **a** (*float*) – Black Hole spin factor
- **Q** (*float*) – Charge on the massive body
- **c** (*float*) – Speed of light
- **G** (*float*) – Gravitational constant
- **Cc** (*float*) – Coulomb's constant

Returns Numpy array of shape (4,4)

Return type array

einsteinpy.utils.kerrnewman_utils.**dmetric_dx**(*r*, *theta*, *M*, *a*, *Q*, *c=299792458.0*, *G=6.67408e-11*, *Cc=8987551787.997911*)

Returns differentiation of each component of Kerr-Newman metric tensor w.r.t. t, r, theta, phi

Parameters

- **r** (*float*) – Distance from the centre
- **theta** (*float*) – Angle from z-axis
- **M** (*float*) – Mass of the massive body
- **a** (*float*) – Black Hole spin factor
- **Q** (*float*) – Charge on the massive body
- **c** (*float*) – Speed of light
- **G** (*float*) – Gravitational constant
- **Cc** (*float*) – Coulomb's constant

Returns **dmdx** – Numpy array of shape (4,4,4) dmdx[0], dmdx[1], dmdx[2] & dmdx[3] is differentiation of metric w.r.t. t, r, theta & phi respectively

Return type array

einsteinpy.utils.kerrnewman_utils.**christoffels**(*r*, *theta*, *M*, *a*, *Q*, *c=299792458.0*, *G=6.67408e-11*, *Cc=8987551787.997911*)

Returns the 3rd rank Tensor containing Christoffel Symbols for Kerr-Newman Metric

Parameters

- **r** (*float*) – Distance from the centre
- **theta** (*float*) – Angle from z-axis
- **M** (*float*) – Mass of the massive body
- **a** (*float*) – Black Hole spin factor
- **Q** (*float*) – Charge on the massive body
- **c** (*float*) – Speed of light
- **G** (*float*) – Gravitational constant
- **Cc** (*float*) – Coulomb's constant

Returns Numpy array of shape (4,4,4)

Return type array

einsteinpy.utils.kerrnewman_utils.**em_potential**(*r*, *theta*, *a*, *Q*, *M*, *c=299792458.0*, *G=6.67408e-11*, *Cc=8987551787.997911*)

Returns a 4-d vector(for each component of 4-d space-time) containing the electromagnetic potential around a Kerr-Newman body

Parameters

- **r** (*float*) – Distance from the centre
- **theta** (*float*) – Angle from z-axis
- **a** (*float*) – Black Hole spin factor
- **Q** (*float*) – Charge on the massive body
- **M** (*float*) – Mass of the massive body
- **c** (*float*) – Speed of light
- **G** (*float*) – Gravitational constant
- **Cc** (*float*) – Coulomb's constant

Returns Numpy array of shape (4,)

Return type array

einsteinpy.utils.kerrnewman_utils.**maxwell_tensor_covariant**(*r*, *theta*, *a*, *Q*, *M*, *c=299792458.0*, *G=6.67408e-11*, *Cc=8987551787.997911*)

Returns a 2nd rank Tensor containing Maxwell Tensor with lower indices for Kerr-Newman Metric

Parameters

- **r** (*float*) – Distance from the centre

- **theta** (*float*) – Angle from z-axis
- **a** (*float*) – Black Hole spin factor
- **Q** (*float*) – Charge on the massive body
- **M** (*float*) – Mass of the massive body
- **c** (*float*) – Speed of light
- **G** (*float*) – Gravitational constant
- **Cc** (*float*) – Coulomb's constant

Returns Numpy array of shape (4,4)

Return type array

einsteinpy.utils.kerrnewman_utils.**maxwell_tensor_contravariant**(*r*, *theta*, *a*, *Q*, *M*, *c=299792458.0*, *G=6.67408e-11*, *Cc=8987551787.997911*)

Returns a 2nd rank Tensor containing Maxwell Tensor with upper indices for Kerr-Newman Metric

Parameters

- **r** (*float*) – Distance from the centre
- **theta** (*float*) – Angle from z-axis
- **a** (*float*) – Black Hole spin factor
- **Q** (*float*) – Charge on the massive body
- **M** (*float*) – Mass of the massive body
- **c** (*float*) – Speed of light
- **G** (*float*) – Gravitational constant
- **Cc** (*float*) – Coulomb's constant

Returns Numpy array of shape (4,4)

Return type array

einsteinpy.utils.kerrnewman_utils.**kerrnewman_time_velocity**(*pos_vec*, *vel_vec*, *mass*, *a*, *Q*)

Velocity of coordinate time wrt proper metric

Parameters

- **pos_vector** (*array*) – Vector with r, theta, phi components in SI units
- **vel_vector** (*array*) – Vector with velocities of r, theta, phi components in SI units
- **mass** (*kg*) – Mass of the body
- **a** (*float*) – Any constant
- **Q** (*C*) – Charge on the massive body

Returns Velocity of time

Return type one

1.7.6 Plotting module

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

Static Geodesic Plotting

This module contains the methods for static geodesic plotting.

class einsteinpy.plotting.senile.geodesics_static.**StaticGeodesicPlotter**(*time=<Quantity 0. s>*, *ax=None*, *attractor_radius_scale=-1.0*, *attractor_color='#ffcc00'*)

Class for plotting static matplotlib plots and animations.

Constructor.

Parameters

- **time** (*Quantity*) – Time of start, defaults to 0 seconds.
- **attractor_radius_scale** (*float, optional*) – Scales the attractor radius by the value given. Default is 1. It is used to make plots look more clear if needed.
- **attractor_color** (*string, optional*) – Color which is used to denote the attractor. Defaults to #ffcc00.

plot_trajectory(*geodesic*, *color*, *only_points=False*)

Parameters

- **geodesic** (Geodesic) – Geodesic of the body.
- **color** (*string*) – Color of the Geodesic

plot(*geodesic*, *color='#e4d8be'*)

Parameters

- **geodesic** (Geodesic) – Geodesic of the body
- **color** (*hex code RGB, optional*) – Color of the dashed lines. Picks a random color by default.

Returns **lines** – A list of Line2D objects representing the plotted data.

Return type list

animate(*geodesic*, *color='#72aa42'*, *interval=50*)

Parameters

- **geodesic** (Geodesic) – Geodesic of the body.
- **color** (*hex code RGB, optional*) – Color of the dashed lines. Picks a random color by default.
- **interval** (*int, optional*) – Control the time between frames. Add time in milliseconds.

Auto and Manual scaling

EinsteinPy supports Automatic and Manual scaling of the attractor to make plots look better since radius of attractor can be really small and not visible.

- **Manual_Scaling :** If the user provides the attractor_radius_scale, then the autoscaling will not work. This is checked by initialising the attractor_radius_scale by -1 and if the user enters the value then it will be >0 so the value won't remain -1 which is easily checked.

  The radius is multiplied to the value given in attractor_radius_scale

```
radius = radius * self.attractor_radius_scale
```

- **Auto Scaling :** If the user does not provide the attractor_radius_scale, the value will be initialised to -1 and then we will call the auto scaling function. In autoscaling, the attractor radius is first initialised to the minimum distance between the attractor and the object moving around it. Now, if this radius is greater than the 1/12th of minimum of range of X and Y coordinates then, the radius is initialised to this minimum. This is done so that the plots are easy to look at.

  minrad_nooverlap : Stores the minimum distance between the particle and attractor .. code-block:: python

  for i in range(0, len(self.xarr)): minrad_nooverlap = min(minrad_nooverlap, self.mindist(self.xarr[i], self.yarr[i]))

  minlen_plot : Stores the minimum of range of X and Y axis

```
xlen = max(self.xarr) - min(self.xarr)
ylen = max(self.yarr) - min(self.yarr)
minlen_plot = min(xlen, ylen)
```

  multiplier : Stores the value which is multiplied to the radius to make it 1/12th of the minlenplot

```
mulitplier = minlen_plot / (12 * radius)
min_radius = radius * mulitplier
```

Attractor Color

- **Color Options :** User can give the color to attractor of his/her choice. It can be passed while calling the geodesics_static class. Default color of attractor is "black".

```
self.attractor_color = attractor_color
mpl.patches.Circle( (0, 0), radius.value, lw=0, color=self.attractor_color)
```

Static Geodesic Plotting (Scatter Plots)

This module contains the basic classes for static plottings in 2-dimensions for scatter and line:

Color

- **Attractor :** User can give the color to attractor of his/her choice. It can be passed while making the object of geodesics_static class. Default color of attractor is "black".

```
self.attractor_color = attractor_color
plt.scatter(0, 0, color=self.attractor_color)
```

- **Geodesic :** User can give the color to the orbit of the particle moving around the attractor of his/her choice. It can be passed while making the object of geodesics_scatter class. Default color is "Oranges".

```
self.cmap_color = cmap_color
plt.scatter(pos_x, pos_y, s=1, c=time, cmap=self.cmap_color)
```

This module contains the methods for static geodesic plotting using scatter plots.

class einsteinpy.plotting.senile.geodesics_scatter.**ScatterGeodesicPlotter**(*time=<Quantity 0. s>*, *attractor_color='black'*, *cmap_color='Oranges'*)

Class for plotting static matplotlib plots.

Constructor.

Parameters

- **time** (*Quantity*) – Time of start, defaults to 0 seconds.
- **attractor_color** (*string, optional*) – Color which is used to denote the attractor. Defaults to black.
- **cmap_color** (*string, optional*) – Color used in function plot.

plot(*geodesic*)

Parameters **geodesic** (Geodesic) – Geodesic of the body

animate(*geodesic*, *interval=50*)

Function to generate animated plots of geodesics.

Parameters

- **geodesic** (Geodesic) – Geodesic of the body
- **interval** (*int, optional*) – Control the time between frames. Add time in milliseconds.

1.7.7 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**(*x*, *y*, *z*)

Class for Cartesian Coordinates and related transformations.

Constructor.

Parameters

- **x** (*Quantity*) –
- **y** (*Quantity*) –
- **z** (*Qauntity*) –

si_values()
Function for returning values in SI units.

Returns Array containing values in SI units (m, m, m)

Return type ndarray

norm()
Function for finding euclidean norm of a vector.

Returns Euclidean norm with units.

Return type Quantity

dot(*target*)
Dot product of two vectors.

Parameters **target** (*Cartesian*) –

Returns Dot product with units

Return type Quantity

to_spherical()
Method for conversion to spherical coordinates.

Returns Spherical representation of the Cartesian Coordinates.

Return type *Spherical*

to_bl(*a*)
Method for conversion to boyer-lindquist coordinates.

Parameters **a** (*Quantity*) – a = J/Mc , the angular momentum per unit mass of the black hole per speed of light.

Returns BL representation of the Cartesian Coordinates.

Return type *BoyerLindquist*

class einsteinpy.coordinates.core.**Spherical**(*r*, *theta*, *phi*)
Class for Spherical Coordinates and related transformations.

Constructor.

Parameters

- **r** (*Quantity*) –
- **theta** (*Quantity*) –
- **phi** (*Quantity*) –

si_values()
Function for returning values in SI units.

Returns Array containing values in SI units (m, rad, rad)

Return type ndarray

to_cartesian()
Method for conversion to cartesian coordinates.

Returns Cartesian representation of the Spherical Coordinates.

Return type *Cartesian*

to_bl(*a*)
Method for conversion to boyer-lindquist coordinates.

Parameters **a** (*Quantity*) – a = J/Mc , the angular momentum per unit mass of the black hole per speed of light.

Returns BL representation of the Spherical Coordinates.

Return type *BoyerLindquist*

class einsteinpy.coordinates.core.**BoyerLindquist**(*r*, *theta*, *phi*, *a*)
Class for Spherical Coordinates and related transformations.

Constructor.

Parameters

- **r** (*Quantity*) –
- **theta** (*Quantity*) –
- **phi** (*Quantity*) –
- **a** (*Quantity*) –

si_values()
Function for returning values in SI units.

Returns Array containing values in SI units (m, rad, rad)

Return type ndarray

to_cartesian()
Method for conversion to cartesian coordinates.

Returns Cartesian representation of the BL Coordinates.

Return type *Cartesian*

to_spherical()
Method for conversion to spherical coordinates.

Returns Spherical representation of the BL Coordinates.

Return type *Spherical*

velocity module

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

class einsteinpy.coordinates.velocity.**CartesianDifferential**(*x*, *y*, *z*, *v_x*, *v_y*, *v_z*)
Class for calculating and transforming the velocity in Cartesian coordinates.

Constructor.

Parameters

- **x** (*Quantity*) –
- **y** (*Quantity*) –
- **z** (*Quantity*) –
- **v_x** (*Quantity*) –
- **v_y** (*Quantity*) –

- **v_z** (*Quantity*) –

si_values()
Function for returning values in SI units.

Returns Array containing values in SI units (m, m, m, m/s, m/s, m/s)

Return type ndarray

velocities(*return_np=False*)
Function for returning velocity.

Parameters **return_np** (*bool*) – True for numpy array with SI values, False for list with astropy units. Defaults to False

Returns Array or list containing velocity.

Return type ndarray or list

spherical_differential()
Function to convert velocity to spherical coordinates velocity

Returns Spherical representation of the velocity in Cartesian Coordinates.

Return type *SphericalDifferential*

bl_differential(*a*)
Function to convert velocity to Boyer-Lindquist coordinates

Parameters **a** (*Quantity*) – a = J/Mc , the angular momentum per unit mass of the black hole per speed of light.

Returns Boyer-Lindquist representation of the velocity in Cartesian Coordinates.

Return type *BoyerLindquistDifferential*

class einsteinpy.coordinates.velocity.**SphericalDifferential**(*r*, *theta*, *phi*, *v_r*, *v_t*, *v_p*)
Class for calculating and transforming the velocity in Spherical coordinates.

Constructor.

Parameters

- **r** (*Quantity*) –
- **theta** (*Quantity*) –
- **phi** (*Quantity*) –
- **v_r** (*Quantity*) –
- **v_t** (*Quantity*) –
- **v_p** (*Quantity*) –

si_values()
Function for returning values in SI units.

Returns Array containing values in SI units (m, rad, rad, m/s, rad/s, rad/s)

Return type ndarray

velocities(*return_np=False*)
Function for returning velocity.

Parameters **return_np** (*bool*) – True for numpy array with SI values, False for list with astropy units. Defaults to False

Returns Array or list containing velocity.

Return type ndarray or list

cartesian_differential()
Function to convert velocity to cartesian coordinates

Returns Cartesian representation of the velocity in Spherical Coordinates.

Return type *CartesianDifferential*

bl_differential(*a*)
Function to convert velocity to Boyer-Lindquist coordinates

Parameters **a** (*Quantity*) – a = J/Mc , the angular momentum per unit mass of the black hole per speed of light.

Returns Boyer-Lindquist representation of the velocity in Spherical Coordinates.

Return type *BoyerLindquistDifferential*

class einsteinpy.coordinates.velocity.**BoyerLindquistDifferential**(*r*, *theta*, *phi*, *v_r*, *v_t*, *v_p*, *a*)
Class for calculating and transforming the velocity in Boyer-Lindquist coordinates

Constructor.

Parameters

- **r** (*Quantity*) –
- **theta** (*Quantity*) –
- **phi** (*Quantity*) –
- **v_r** (*Quantity*) –
- **v_t** (*Quantity*) –
- **v_p** (*Quantity*) –
- **a** (*Quantity*) –

si_values()
Function for returning values in SI units.

Returns Array containing values in SI units (m, rad, rad, m/s, rad/s, rad/s)

Return type ndarray

velocities(*return_np=False*)
Function for returning velocity.

Parameters **return_np** (*bool*) – True for numpy array with SI values, False for list with astropy units. Defaults to False

Returns Array or list containing velocity.

Return type ndarray or list

cartesian_differential()
Function to convert velocity to cartesian coordinates

Returns Cartesian representation of the velocity in Boyer-Lindquist Coordinates.

Return type *CartesianDifferential*

spherical_differential()
Function to convert velocity to spherical coordinates

Returns Spherical representation of the velocity in Boyer-Lindquist Coordinates.

Return type *SphericalDifferential*

1.7.8 Constant module

1.7.9 Units module

einsteinpy.units.**astro_dist**(*mass*)
Function for turning distance into astronomical perspective.

einsteinpy.units.**astro_sec**(*mass*)
Function for turning time into astronomical perspective.

1.7.10 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>*, *R=<Quantity 0. km>*, *differential=None*, *a=<Quantity 0. m>*, *q=<Quantity 0. C>*, *parent=None*)
Class to create a generic Body

Parameters

- **name** (*str*) – Name/ID of the body
- **mass** (*kg*) – Mass of the body
- **R** (*units*) – Radius of the body
- **differential** (*coordinates, optional*) – Complete coordinates of the body
- **a** (*m, optional*) – Spin factor of massive body. Should be less than half of schwarzschild radius.
- **q** (*C, optional*) – Charge on the massive body
- **is_attractor** (*Bool, optional*) – To denote is this body is acting as attractor or not
- **parent** (Body, *optional*) – The parent object of the body.

1.7.11 Geodesic module

class einsteinpy.geodesic.**Geodesic**(*body*, *end_lambda*, *step_size=0.001*, *time=<Quantity 0. s>*, *metric=<class 'einsteinpy.metric.schwarzschild.Schwarzschild'>*)
Class for defining geodesics of different geometries.

PYTHON MODULE INDEX

e

INDEX

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V

W