Draft version January 22, 2020
Typeset using LaTeX default style in AASTeX63

EinsteinPy: Python for General Relativity*

Shreyas Bapat,[1,†] Shilpi Jain,[2,‡] Ritwik Saha,[1,†] Bhavya Bhatt,[1,§] Akshita Jain,[1]

(Primary Paper Contributors)

Priyanshu Khandelwal,[1,‡] Sofía Ortín Vela,[3] Jialin Ma,[4] Varun Singh,[1] Alpesh Jamgade,[5] Manvi Gupta,[1] Tushar Tyagi,[1] Tanmay Rustagi,[1] Abhijeet Manhas,[1] Raphael Reyna,[6] Sashank Mishra,[7]

(EinsteinPy Contributors)

[1] *School of Computing and Electrical Engineering, Indian Institute of Technology Mandi, India*
[2] *Department of Earth Sciences, Indian Institute of Technology Roorkee, India*
[3] *University of Zaragoza, Spain*
[4] *Georgia Institute of Technology, USA*
[5] *Department of Mathematics, Bharath Institute of Higher Education and Research, Chennai, India*
[6] *California State Polytechnic University, Pomona*
[7] *Department of Information Technology, Indian Institute of Information Technology Allahabad*

(Received June 1, 2019; Revised January 10, 2019; Accepted January 22, 2020)

Submitted to ApJ

ABSTRACT

This paper presents EinsteinPy (version 0.3), a community-developed Python package for gravitational and relativistic astrophysics. Python is a free, easy to use high level programming language which has seen a huge expansion in the number of its users and developers in recent years. Specifically, a lot of recent studies show that the use of Python in Astrophysics and in general physics has increased exponentially. Many great frameworks came as Python packages which provide a very high level of abstraction over the dirty nitty-gritty of complex algorithms and provide an easy to use interfaceand pleasing user experience. One such example is Keras - framework for deep learning which has made deep learning so easy that a person with zero programming knowledge can also train a neural network classifier. This example really demonstrates the power of abstraction which is achievable in Python. The aim of the EinsteinPy is no different and is developed keeping in mind the state of a theoretical gravitational physicist with a little or no background in computer programming and trying to work in the field of numerical relativity or trying to use simulations in their research. Currently EinsteinPy supports simulation of time-like and null geodesics and calculate trajectories in different background geometries some of which are Schwarzschild, Kerr and KerrNewmann along with coordinate inter-conversion pipeline. It has a partially developed pipeline for plotting and visualization with dependencies on libraries like plotly, matplotlib etc. One of the unique feature of EinsteinPy is a sufficiently developed symbolic tensor manipulation utilites which is a great tool in itself for teaching yourself tensor algebra which for many beginner students can be overwhelmingly tricky. Currently EinsteinPy also provide few utility functions for hypersurface embedding of Schwarzschild spacetime which further will be extended to model gravitational lensing simulation. The current version of the library is in a state that can be used by any serious student of general relativity trying to get essence of this beautiful subject but is somewhere lost in the heavy mathematical formalism of the subject.

Corresponding author: Shreyas Bapat
shreyas@einsteinpy.org, bapat.shreyas@gmail.com

* Released on July, 15th, 2019

EinsteinPy provides such students to really see through the equations and visualize whats really happening.

Keywords: gravitational physics, astrophysics — simulations — black holes — gravitational waves

1. INTRODUCTION

The relativistic theory of gravity paved its way to the ground in 1915?, when Prof. Albert Einstein published his paper on the general theory of relativity. This elegant and rigorous framework was a generalized version of gravity-free theory - special relativity, which he published earlier in 1905. Almost a hundred years after Einstein wrote down the equations of General Relativity, solutions of the Einstein field equations remain extremely difficult to find beyond those which exhibit significant symmetries.

We can study the behavior of solutions under a high degree of symmetry considerations and could even solve analytically for highly unrealistic systems. In case of problems relevant to astrophysical and gravitational physics research, it still remains a big daunting question on how to get around the problem of solving these field equations. This question is so profound that it has a separate profound field of research which goes with the name of numerical relativity which attempts to use computation science and programming to numerically obtain solutions of the equations (which would be the background geometry) for some turbulent region where most of the interesting dynamics are happening (as we can not solve for an infinitely large grid due to restrictions imposed on us by space and time complexity and computability).

Numerical methods have been used to solve the Einstein equation for many decades, but the past decade has seen tremendous advances and adding to that, the interest of the community grew when the field found applications in areas of radio astronomy, cosmology, signal processing, data mining. The field of numerical relativity is growing rampantly with the vast literature on algorithms, numerical methods and theoretical formulations (from basic 3+1 decomposition formulation to more sophisticated ones) with the development of robust and involved frameworks that provide a complete programming ecosystem and have proved to be essential tools for any numerical relativity researcher.

Reflecting the other end of the research community stands the major section of theoretical physicists which counts the majority as a novice in programming. The head challenge is the fact that the usage of these frameworks demands heavy use of high-level programming languages like C, C++, Python, etc. Though Python provides a vast room for abstractions no library existed that focused towards numerical relativity, eventually, the need of python library dedicated to general relativity became the propellant for the team and thus they made dauntless efforts to cover the space for this Python library. Since then, EinsteinPy has seen a lot of contributions from people all over the world and many "good to go" functionalities are already provided in this and previous versions.

EinsteinPy is made to provide a set of tools which can make numerical computations for solving Einstein's field equations an easy job for anyone who does not want to deal with intricacies of the subject along with few, very basic but powerful functionalities that could be used by anyone who wants to learn the subject of general relativity with every minute detail. The library is as an attempt to provide programming and numerical environment for numerous 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.

EinsteinPy provides an open-source and open-development core package and aims to build an ecosystem of affiliated packages that support numeric relativity functionality in the Python programming language. "Open development" describes a process where anybody with an internet connection can suggest changes to the source code and contribute their opinion when new features, bug fixes or other code changes, governance, or any other aspect of the development process is discussed. EinsteinPy core package is now a feature-rich library of sufficiently general tools and classes that support the development of more specialized code

In the further coming section, we discuss with some of the features of the current as well previous version and the future plans which are yet to be implemented in the upcoming versions.Also, we describe a few code snippets to explain the usage of the library on which more details can be found on the organization website. The development of the

† EinsteinPy Lead Developer, EinsteinPy Board Member
‡ EinsteinPy Board Member
§ EinsteinPy Deputy Lead Developer

EinsteinPy core package began as a community-driven effort to standardize core functionality for software in Python which renders a user-friendly interface for supporting numerical relativity and relativistic astrophysics research.
We start this paper by describing the way EinsteinPy functions followed by the main software efforts developed by the EinsteinPy itself with proper detailed documentation of the current version release. We end with a short vision for the future of EinsteinPy to its contribution in the field of computational methodology in gravitational and relativistic Astrophysics.

2. DATA TYPES

The EinsteinPy package provides some core data types for calculating, operating on, visualizing geodesics and black holes. There are some data types created for symbolic manipulations and calculations of relativistic tensors. The most important data types for EinsteinPy are `Body`, `Geodesic` and `BaseRelativityTensor` objects that support defining and creating new bodies, storing and calculating geodesic for bodies, and handling respectively. The main purpose of these core classes is to standardize the way calculations and visualizations are done for various kinds of heavy cosmic bodies and black holes. The classes always maintain the consistency in units of measurement wherever applicable, and create a consistent and clean API for any non-programmer to understand. The classes are designed such that it resembles the flow of physics and are as intuitional as possible. These core classes also include a rigid architecture for data manipulation and visualizations wherever applicable. Following sections provide deep insight into the above mentioned three code data types of EinsteinPy:

2.1. *Body*

The Body data type helps define the central black-hole or the heavy massed object and the objects revolving around it. A new body can be defined by some of the very basic constraints:

- `name`: Body Name,
- `mass`: Mass of the body with attached astropy units
- `R`: Radius of the body with attached astropy units
- `differential`: Complete position and velocity vector of the body in form of a Cartesian / Spherical / Boyer-Lindquist data class object from einsteinpy coordinates (optional)
- `a`: Spin factor (a) of the body for Kerr Solutions with attached astropy units (optional)
- `q`: Charge of the body with attached astropy units (optional)
- `is_attractor`: A boolean variable concerned with if the body is to be supposed as a central attractor. (optional)
- `parent`: Parent body for the body, object of the same data class (optional)

2.2. *Geodesic*

The Body data type helps define the central black-hole or the heavy massed object and the objects revolving around it. A new body can be defined by some of the very basic constraints:

2.3. *BaseRelativityTensor*

This is a base class from which other tensors in *symbolic* module are derived. The user can use this class to create his/her own tensor while using the various in-built functions provided by this class. The class also maintains the indicial configuration of the tensor(contravariant and covariant indices), calculates the free variables used in describing the tensor other than those which depicts the axis of space-time itself (e.g. t, x, y, z). The class has a function to convert the symbolic tensorial quantities into functions where actual numerical values can be substitued.
This class along with its parent class *Tensor* have the following functions/properties,

- `order` (Property) : Returns the order of the tensor.
- `config` (Property) : Returns the arrangement of contravariant(upper)/covariant(lower) indices of the tensor. For example, 'ulll'.

- `parent_metric` (Property) : Returns the metric tensor of the space-time associated with the tensor, if present. Else returns `None`.
- `tensor` (Function) : Returns the tensorial quantity contained within the class instance.
- `subs` (Function) : Returns a new tensor upon substitution of current variables with new variables/values.
- `simplify` (Function) : Returns a tensor contain simplified expressions from the current tensor.
- `symbols` (Function) : Returns the symbols denoting each axis of the space-time.
- `tensor_lambdify` (Function) : Returns lambdified function of symbolic tensors. This means that the returned functions can accept numerical values and return numerical quantities.
- `lorentz_transform` (Function) : Returns a lorentz-transformed tensor when supplied with a transformation matrix. This function also considers the contravariance and covariance of tensors during transformation.

3. UNIT HANDLING

All user-facing functionality provided by einsteinpy make use of astropy.units. All functions and objects must have their input constrained to the appropriate type of unit (e.g., length, mass or energy). Inputs can then be provided with any units that match the required type (e.g., millimeters and inches are both valid units for length) and conversions occur automatically without user intervention. The einsteinpy.constant subpackage contains a few standard constants.

4. DEVELOPMENT MODEL

The EinsteinPy Project follows an open development model, which is widely used in the scientific Python community and other computational astrophysics packages. The project is hosted on GitHub as a public repository with restricted write access to some key people. Anyone, however is welcome and encouraged to make code contributions and suggest changes using pull requests. Since the repository containing the codebase is licenced under an open and permissive MIT licence, the project allows commercial use, modification, sublicensing, distribution and private use as long as the copyright notice and licence is redistributed. The project uses git for distributed version control and relies on CircleCI, Codefactor, GitHub Actions etc. for running the unit tests and other CI checks for every pull request. The code quality is maintained at the highest standards with strong control on the cyclomatic complexity, code linting, imports ordering and a proper docstring wherever necessary. Every contribution is reviewed and it must meet the following requirements:

- The code must strictly adhere to the PEP8 Standards, on the top of it, the code must be linted with black to save the reviewer's time identifying the changes made and reduce the diff generated because of the patch.
- The imports in the code must be sorted using tools such as isort.
- The documentation follows numpy like docstrings, and a proper reStructured Text syntax is followed for writing the documentation.
- The changelog is populated at release-time by the lead developer and the releasing developer

EinsteinPy further follows a trunk-based development model, which ensures a new branch for every minor release and new patch release made using the minor release branch. All the latest changes are kept on the master branch on GitHub and the relevant issues are closed as soon as the pull request solving it merges onto the master. A release having the changes is made at a later date. EinsteinPy only supports the latest bleeding-edge Python, Numpy and Astropy versions. All the stable Python versions after Python 3.5 are supported. All the code snippets written follow the guidelines of numba in order to accelerate the Python code and make the simulations faster.

5. THE EINSTEINPY CORE PACKAGE

This section explains in detail, the modules available within EinsteinPy core pacakge, their API and the purpose they serve.

5.1. *METRIC*

This module captures all the geometric and causal structure of specific spacetimes, and being able to calculate trajectories in a given space-time. The module derives its name from Metric Tensor, a tensorial quantity used to describe differential lengths, especially in a curved space-time. However, this module serves more than its usual definition and ultimately leads us to obtain geodesics of particles in a spacetime with high curvature, where notions of newtonian physics fail.

5.1.1. *Schwarzschild Metric*

Karl Schwarzchild pioneered the analysis of the relation between the size of black hole and its mass and this work paved the way for the first exact solution of Einstein's Field Equation under the limited case of single spherical non-rotating, non-charged body. The metric equation corrosponding to a space-time centred around a body in the above described conditions is given by,

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

The same equation can be written in a tensorial form as follows,

$$g_{\mu\nu} = \begin{bmatrix} -(1 - \frac{r_s}{r})c^2 & 0 & 0 & 0 \\ 0 & \frac{1}{(1 - \frac{r_s}{r})} & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2 sin^2\theta \end{bmatrix} \tag{2}$$

$$ds^2 = g_{\mu\nu} dx^\mu dx^\nu \tag{3}$$

where r_s is the schwarzschild radius of a given heavy body and is given by the equation,

$$r_s = \frac{2GM}{c^2} \tag{4}$$

G, M, and c are gravitational constant, mass of the body and speed of light in vacuum respectively.

Schwarzschild Radius is a limit below which the degree of space-time curvature causes the partcle to undergo irreversible gravitational collapse wherein the partcle would need a speed greater than speed of light ot escape singularity.

As $r \to \infty$, the metric defined above starts to approximate flat Minkowski space-time, thus showing that Schwarzschild geometry is asymptotically flat.

If supplied with initial position and velocity of a particle with negligible mass(in comparision to the central body), we can obtain the geodesics of the particle for a given range of proper time τ, as it boils down to a set of eight differential equations numerically solvable by a class of Runge-Kutta methods.

The file **schwarzschild.py** contains the class *Schwarzschild* for defining Schwarzschild geometry methods.

Here is a working code describing the motion of particle revolving around a black hole of mass approximately equal to mass of the Earth. As the trajectory starts from a mere $130m$ from the black hole, we obtain some interesting observations.

Listing 1. Importing the required modules

```
>>> import numpy as np
>>> import astropy.units as u
>>> from einsteinpy.plotting import GeodesicPlotter
>>> from einsteinpy.coordinates import SphericalDifferential
>>> from einsteinpy.bodies import Body
>>> from einsteinpy.geodesic import Geodesic
>>> from einsteinpy.metric import Schwarzschild
```

Listing 2. Defining the attractor and the particle and calculating geodesics

```
>>> Attractor = Body(name="BH", mass=6e24 * u.kg)
>>> init_conditions = 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)
>>> Particle = Body(differential=init_conditions, parent=Attractor)
>>> geodesic = Geodesic(body=Particle, time=0 * u.s, end_lambda=0.00205, step_size=5e-8,
...                     metric=Schwarzschild)
>>> geodesic.trajectory
array([[ 0.00000000e+00,  1.20104339e+02, -4.97488462e+01, ...,
         9.45228078e+04,  2.28198245e+05,  0.00000000e+00],
       [ 3.99986427e-08,  1.20108103e+02, -4.97397110e+01, ...,
         9.36472607e+04,  2.28560869e+05, -5.80280784e-14],
       ...,
       [ 2.04962579e-03,  1.04114904e+02,  1.08255773e+01, ...,
        -1.23245830e+06,  1.80261847e+05, -9.67931265e-09],
       [ 2.05002578e-03,  1.03619005e+02,  1.08973776e+01, ...,
        -1.24706334e+06,  1.78734562e+05, -9.79337222e-09]])
>>> geodesic.trajectory.shape
(6567, 8)
```

We can see that for each timestep, we obtain eight values : t, r, θ, ϕ, v_t, v_r, v_θ and v_ϕ corrosponding to coordinate time, radial distance from the centre, polar angle, azimuth angle and their respective velocities w.r.t proper time τ.

Further, we plot the trajectory in 2D for vizualization purposes.

Listing 3. Plotting the trajectory

```
>>> obj = GeodesicPlotter()
>>> obj.plot(geodesic)
>>> obj.show()
```

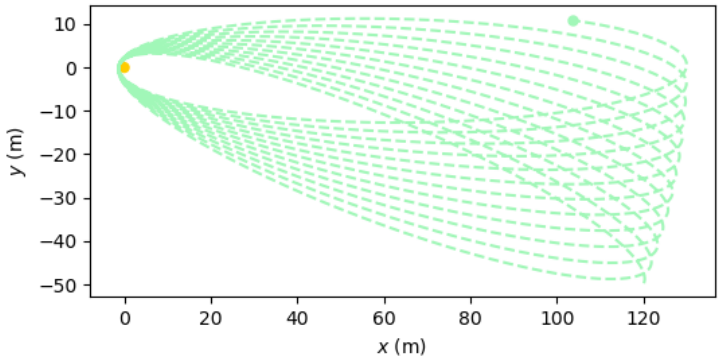

Figure 1. Plot obtained from the above code showing the phenomenon of perihelion advance

We observe the phenomenon of perihelion advance of a particle around a heavy body due to spacetime curavture. The orbit would have been purely eliptical if plotted in accordance with Newtonian Mechanics, however, the curvature of space-time around the heavy mass shifts the orbit with each revolution. In practicality, this phenomenon is not so pronounced as no particle approaches a heavy mass this closely. For instance, Mercury advances its perihelion at a rate of 42.98 ± 0.04 arcsec/century.

5.1.2. *Kerr Metric*

Kerr metric is a further generalization of Schwarzschild metric and a specific case of Kerr-Newman geometry. A spacetime is defined as a "Kerr" spacetime as and when the massive central body posseses an angular momentum. The corresponding metric equation which defines the space-time is,

$$g_{\mu\nu} = \begin{bmatrix} -(1-\frac{r_s r}{\Sigma})c^2 & 0 & 0 & -\frac{r_s r a sin^2\theta}{\Sigma}c \\ 0 & \frac{\Sigma}{\Delta} & 0 & 0 \\ 0 & 0 & \Sigma & 0 \\ -\frac{r_s r a sin^2\theta}{\Sigma}c & 0 & 0 & (r^2 + a^2 + \frac{r_s r a^2}{\Sigma} sin^2\theta) sin^2\theta \end{bmatrix} \tag{5}$$

$$ds^2 = g_{\mu\nu} dx^\mu dx^\nu \tag{6}$$

where,

$$a = \frac{J}{Mc},$$
$$\Sigma = r^2 + a^2 cos^2, \theta$$
$$\Delta = r^2 - r_s r + a^2$$

J is the angular momentum of the body and all the other symbols hold their usual meanings as discussed in 5.1.1. The coordinate system used to describe the metric is Boyer-Lindquist coordinate system.

The file **kerr.py** contains the class *Kerr* for defining Schwarzschild geometry methods.

APIs for *Schwarzschild*, *Kerr* and *KerrNewman* are consistent and code structure remains same for the sake for clarity and intuitiveness. For example,

Listing 4. Defining the Kerr Body and the particle and calculating geodesics

```
>>> a = 0.3 * u.m
>>> Attractor = Body(name="BH", mass=1.989e30 * u.kg, a=a)
>>> init_conditions = BoyerLindquistDifferential(49.95e5 * 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)
>>> Particle = Body(differential=init_conditions, parent=Attractor)
>>> geodesic = Geodesic(body=Particle, time=0 * u.s, end_lambda=33932.90,
...                     step_size=1.2, metric=Kerr)
```

As evident from the code, API across different classes are made to be as homegenous as possible.

As a rotating black hole drags the space-time around it, a freely falling object acquires a angular velocity around the coordinate centre, i.e centre of the black hole. This effect is evident from the plot obatined from the above geodesics, in the same way as obtained one in the Schwarzschild example.

Figure 2. Plot obtained from the above geodesics showing the phenomenon of frame dragging

5.1.3. *Kerr-Newman Metric*

Kerr-Newman metric presents the most general case within single-body vacuum solutions of Einstein Field Equation. It generalizes all the quantities used to describe a black hole, as affirmed by "no hair" theoram, i.e. Charge Q, Angular Momentum J and Mass M. The metric equation is given by,

$$-ds^2 = c^2 d\tau^2 = -(\frac{dr^2}{\Delta} + d\theta^2)\rho^2 + (cdt - asin^2\theta d\phi)^2 \frac{\Delta}{\rho^2} - ((r^2 + a^2)d\phi - acdt)^2 \frac{sin^2\theta}{\rho^2} \tag{7}$$

where,

$$a = \frac{J}{Mc},$$
$$\Sigma = r^2 + a^2 cos^2\theta,$$
$$\Delta = r^2 - r_s r + a^2 + r_Q^2,$$
$$r_Q^2 = \frac{Q^2 G}{4\pi\epsilon_0 c^4}$$

All the other symbols convey their usual meanings, as discussed in the previous two sections 5.1.1 and 5.1.2. For the sake of brevity, we would not go into the specifics of this class as the classes within *Metric* module share very similar APIs as inferred in 5.1.2.

5.2. *COORDINATES*

5.3. *SYMBOLIC*

General Relativity required heavy mathematical formulation to derive and describe various tensors attributing to various features essential for cogent description of an arbitrary space-time. We would describe the various quantities currently supported by this module while maintaining brevity. Each quantity has its own class, even scalars, which are treated as 0^{th} order tensors.

All the mathematical quantities build upon *BaseRelativityTensor* as derived classes, which is discussed in detail in 2.3.

5.3.1. *MetricTensor*

Metric Tensor describes the differential length elements required to measure distance in a curved(also applies to flat) space-time. It is a second order tensor and is fundamental to describe any space-time geometry. The *MetricTensor* class, being inherited from *BaseRelativityTensor* , inherits all its functions and also have some functions and constraints of its own. For example, metric tensor is bound to be a second order tensor.

The defined class functions are :

- `change_config` : returns a different instance of the same class with different index (contravariant or covariant) configuration.
- `inv` : returns inverse of the metric

Also, the library maintains an ever expanding predefined metrics for direct use. Here's an code snippet for a better dive into the specificities of the class.

Listing 5. Importing a predefined metric

```
>>> from einsteinpy.symbolic.predefined import CMetric
>>> metric = CMetric
>>> metric.tensor()
```

$$\begin{bmatrix} -\frac{h(y)}{(x+y)^2} & 0 & 0 & 0 \\ 0 & \frac{1}{(x+y)^2 f(x)} & 0 & 0 \\ 0 & 0 & \frac{1}{(x+y)^2 h(y)} & 0 \\ 0 & 0 & 0 & \frac{f(x)}{(x+y)^2} \end{bmatrix}$$

Listing 6. Displaying the contravarint configuration of metric

```
>>> metric.change_config("uu").tensor()
```

$$\begin{bmatrix} -\frac{(x+y)^2}{h(y)} & 0 & 0 & 0 \\ 0 & (x+y)^2 f(x) & 0 & 0 \\ 0 & 0 & (x+y)^2 h(y) & 0 \\ 0 & 0 & 0 & \frac{(x+y)^2}{f(x)} \end{bmatrix}$$

5.3.2. *ChristoffelSymbols*

Although Christoffel Symbols are belong to class of pseudo-tensors, it is inherited from *BaseRelativityTensor* to maintain homogenity within the module. Being inherited from *BaseRelativityTensor*, it already supports the functions of its parent class and imposes some restrictions of its own, such as order of the tensor should be 3.

Instance of the class can be made by directly passing a 3^{rd} order tensor or can be obtained using the following classmethods,

- `from_metric` : Calculates christoffel symbols from a given instance of *MetricTensor* class.

The defined class function(s) are :

- `change_config` : returns a different instance of the same class with different index (contravariant or covariant) configuration.

5.3.3. *RiemannCurvatureTensor*

This quantity is the most extensive measure of curvature of a sapce-time. It is inherited from *BaseRelativityTensor* and supports all the functions of its parent class.

Instance of the class can be made by directly passing a 4^{th} order tensor or can be obtained using the following classmethods

- `from_metric` : Calculates Riemann Tensor from a given instance of *MetricTensor* class.
- `from_christoffels` : Calculates Riemann Tensor from a given instance of *ChristoffelSymbols* class.

The defined class function(s) are :

- `change_config` : returns a different instance of the same class with different index (contravariant or covariant) configuration.

5.3.4. *RicciTensor*

This tensor is used in solving Eintein's equation and basically contraction of Riemann Curvature Tensor. It is inherited from *BaseRelativityTensor* and supports all the functions of its parent class.

Instance of the class can be made by directly passing a 2^{nd} order tensor or can be obtained using the following classmethods

- `from_metric` : Calculates Ricci Tensor from a given instance of *MetricTensor* class.
- `from_christoffels` : Calculates Ricci Tensor from a given instance of *ChristoffelSymbols* class.
- `from_riemann` : Calculates Ricci Tensor from a given instance of *RiemannCurvatureTensor* class.

The defined class function(s) are :

- `change_config` : returns a different instance of the same class with different index (contravariant or covariant) configuration.

5.3.5. *RicciScalar*

It is a scalar quantity obtained by contracting a given Ricci Tensor. It is inherited from *BaseRelativityTensor* and supports all the functions of its parent class. Being a scalar quantity, it does not support `change_config` unlike other tensors as discussed above.

Instance of the class can be made by directly passing a symbolic expression or can be obtained using the following classmethods

- `from_metric` : Calculates Ricci Scalar from a given instance of *MetricTensor* class.
- `from_christoffels` : Calculates Ricci Scalar from a given instance of *ChristoffelSymbols* class.
- `from_riemann` : Calculates Ricci Scalar from a given instance of *RiemannCurvatureTensor* class.
- `from_riccitensor` : Calculates Ricci Scalar from a given instance of *RicciTensor* class.

The defined class properties are :

- `expr` : returns the scalar symbolic expression contained in the class.

Here is a sample code snippet, using the above discussed Classes

Listing 7. Importing Anti De-Sitter metric

```
>>> from einsteinpy.symbolic.predefined import AntiDeSitter
>>> from einsteinpy.symbolic import RicciTensor, RicciScalar
>>> import sympy
```

Listing 8. Obtaining Ricci Tensor from the metric

```
>>> RT = RicciTensor.from_metric(AntiDeSitter())
>>> RT.tensor()
```

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

Listing 9. Deriving Ricci(Curvature) Scalar from Ricci Tensor

```
>>> RS = RicciScalar.from_riccitensor(RT)
>>> sympy.simplify(RS.expr)  # simplifying the expression
```

$$-12$$

In congruence to theoretical results, the code outputs a constant negetive scalar curvature for Anti De-Sitter spacetime. All the above discussed classes are called in this snippet as *RicciTensor* internally calls *RiemannCurvatureTensor*, which in turn calls *ChristoffelSymbols*.

5.3.6. *Other Defined Quantities*

As the library is in heavy development, new quantities are being added as and when required. Some other quantities currently supported are listed below. The quantities are all inherited from *BaseRelativityTensor*, and a similar API (same as that of discussed above) has been implemented to maintain compatiblity and homegenity. The class names denote usual quantities relevant to General Relativity.

- *EinsteinTensor*
- *StressEnergyMomentumTensor*
- *WeylTensor*
- *SchoutenTensor*

5.4. *HYPERSURFACE*

5.5. *PLOTTING*

6. INFRASTRUCTURE

6.1. *Testing*

The EinsteinPy Project maintains a very high bar in accepting changes and introducing new features. Every patch of code that is accepted in the codebase through any pull request has to be well tested using some unit tests unless given an exception after discussion with both the lead developers. Currently the coverage of unit tests, which means the total fraction of codebase run by at least one unit test is 94% and we hope to improve it in the future. The 100% code coverage for this software at this level is not possible because of lack of accurate data to test the code against. We deliberately leave some parts of the codebase out of the testing loop because testing those modules is not possible with the current knowledge level and expertise. However, 94% code coverage is good enough when we consider other packages in scientific Python community who face the similar problem.

The test suite can be run manually as well as it runs in an automated fashion on every commit as well as pull request to maintain the code standards, and make it easier for contributors to understand the implications and impact of their changes. The EinsteinPy Project uses many continuous-integration services which provide services to create automated pipelines for code inspection, unit testing, coverage publishing etc. All the contributions trigger these free services (Codecov, CircleCI, Appveyor, GitHub Actions, Codefactor, Codeclimate, Hound, WIP) which are integrated with GitHub repository. All the pull requests are checked for code linting by a pep8speaks bot, which suggests the lines of code which violate the PEP8 standard to the contributor in a human readable way. All these services run on three types of operating systems (Windows, MacOS and Linux). The unit tests run on various versions of Python. They test the documentation builds, test coverage, check whether the pull request author has indicated that it is a work in progress (WIP).They also run several code linting checks which further triggers the unit test build.

6.2. *Documentation and gallery*

7. COMMUNITY

EinsteinPy is being developed as an open source community inside the wide and diverse general scientific python community. Our code is freely available, and we have seen a lot of people contributing and joining us since over the past year. We touched the landmark of 30 members recently. EinsteinPy is dedicated to problems arising in General Relativity and gravitational physics as well as encouraging open source development. EinsteinPy has benefited greatly from summer of code schemes. Last year EinsteinPy participated in the ESA Summer of Code In Space (SOCIS). This program is inspired by Google Summer Of Code (GSOC) and aims to raise the awareness of open source projects related to space, promote the European Space Agency. The symbolic module in EinsteinPy project was developed during this program.

8. FUTURE

We intend to add more functionalities to our project, some of which are,

- Support for null-geodesics in different geometries, .
- Relativistic hydrodynamics
- Adaptive Mesh Refinement
- Providing numerical solutions to Einstein's equations for arbitrarily complex matter distribution.

We have an open community where suggestions from people to add functionalities are welcome. We wish to be as helpful as possible to people using EinsteinPy in.

ACKNOWLEDGMENTS

We thank all the people that have made this AASTeX what it is today. This includes but not limited to Bob Hanisch, Chris Biemesderfer, Lee Brotzman, Pierre Landau, Arthur Ogawa, Maxim Markevitch, Alexey Vikhlinin and Amy Hendrickson. Also special thanks to David Hogg and Daniel Foreman-Mackey for the new "modern" style design. Considerable help was provided via bug reports and hacks from numerous people including Patricio Cubillos, Alex Drlica-Wagner, Sean Lake, Michele Bannister, Peter Williams, and Jonathan Gagne.

Facilities: HST(STIS), Swift(XRT and UVOT), AAVSO, CTIO:1.3m, CTIO:1.5m,CXO

Software: astropy (**?**), Cloudy (**?**), SExtractor (**?**)

APPENDIX

A. APPENDIX INFORMATION

Appendices can be broken into separate sections just like in the main text. The only difference is that each appendix section is indexed by a letter (A, B, C, etc.) instead of a number. Likewise numbered equations have the section letter appended. Here is an equation as an example.

$$I = \frac{1}{1 + d_1^{P(1+d_2)}} \tag{A1}$$

Appendix tables and figures should not be numbered like equations. Instead they should continue the sequence from the main article body.