

# Jupyter Notebook

May 29, 2022

## 1 Open source alternatives for symbolic analysis: GR applications

### 1.1 Introduction

Computer Algebra (CA) systems are computational tools of growing importance in the theoretical physics community. Their application in the context of general relativity is nicely illustrated by [“Overview of Computer Algebra in Relativity”](#) by [D. Hartley](#), from which we extract some key points.

Generally speaking, CA refers to the theory and implementation of computational programs to perform symbolic manipulations as well as calculations typically encountered in mathematics, i.e. algebra, calculus, etc.

The main capabilities offered by most general-purpose CA systems are:

1. Arbitrary precision arithmetic
2. Exact calculation with symbolic mathematical expressions
3. Manipulation of expressions and sub-expressions
4. Analytic and numeric approximation
5. Extension by user programming

One may ask what are the reasons for choosing CA systems. The reasons are quite obvious:

1. Exact results
2. CA systems can perform large tedious calculations much more quickly than is possible by hand
3. In some sense CA systems supplant tables of integrals, ODEs, Fourier transforms, and so on

Along with these basic features for symbolic calculations, CA systems offer additional capabilities for graphics and numerical computation, thereby providing a fast and handy tool to mathematicians and physicists.

Since calculations in General Relativity (GR) are notoriously complicated (i.e. lengthy tensorial and differential calculations), CA methods have shown immensely useful in many research aspects related to GR.

[“Computer algebra in gravity research”](#) by [Malcolm A. H. MacCallum](#) describes the state of the art of CA for GR, as well as its current major implementation used in research. We report here some of the main considerations of this article to better appreciate what we are going to do.

The advantages of CA are as follows:

1. **Accuracy.** CA gives exact and accurate answers: well-written and tested computer algebra packages do not lose factors of 2 or get the sign wrong.
2. **Speed.** CA is fast: the classic example is Delaunay's calculations of lunar motion in the 19th century, which took him 20 years, but was repeated and corrected (using at least 18 h of IBM 360-44 CPU time) by [Deprit et al. \(1970\)](#). An example in relativity is given by the famous Bondi metric, where the initial calculations ([Bondi et al. 1962](#)) took 6 months and still contained errors, whereas d'Inverno's clam could repeat the calculation (correctly!) in 18 s, and nowadays PC now does it in milliseconds.
3. **Repetitive tasks.** CA removes the tediousness of almost identical repeated calculations, for example calculations of curvature for similar metrics.
4. **Calculations infeasible by hand.** The power of the systems opens up projects which would be unthinkable by hand, sometimes with unexpected results. For example, a brute force calculation may lead to a simple answer which prompts new insight, revealing deeper principles and enabling the answer to be derived more elegantly.

Many reasons have to be accounted in order to predict, if a CA package will die or age.

" [...] I want to state firmly that there is no best package, either for CA in general or CA in GR in particular. If you have found a package which works with the machine and operating system, or within the general purpose CA system you are already familiar with, and which does the calculations you want to do, or can readily be adapted to do so, use that. And if you have not found such a package (yet), look for one such rather than seek a -best- one. " by MacCallum

However, one should not forget that software may always contain bugs. Therefore, it is wise to support and maintain several competitive CA implementations to be able to cross check the calculations. Moreover, scientific research funded with public money should not stick to commercial, proprietary software if a valid free alternative exists. In this notebook we, thus, specifically explore open source, community-driven CA implementations.

## 1.2 Goal of this notebook

The aim of this notebook is to compare [Mathematica](#) and free and open-source [Python](#)-based alternatives for symbolic computations and analysis. The main focus is on GR applications. See also “[Symbolic and numerical analysis in general relativity with open source computer algebra systems](#)” (2018), “[Computer algebra in gravity research](#)”(2018) and “[Using Maple and GRTensorIII in relativistic spherical models](#)” for recent explorations of GR symbolic computation using [SageMath](#) and [Maple](#).

First, we generate a test notebook with basic features used in Mathematica and [RGCT](#): <https://faculty.washington.edu/lgy/ph564/>.

We then use two similar open source Python packages that implement the same features using [SymPy](#):

1. [EinsteinPy](#)
2. [GraviPy](#)

We compare the results and give some final judgments.

**Notation:**

We will notice from time to time some differences between Mathematica and the python packages used as follows:

**DIFFERENCE:** *explanation*

In order to see what the specifications of each function are just remove the `#` in front of

```
help( )
```

and it will show you examples and arguments for the function.

## 2 Simple computations and time performances of SymPy

```
sympy==1.10.1
```

### 2.0.1 Analysis

```
[ ]: #clear all variables
      %reset -f
```

**DIFFERENCE:** In *sympy*, one must explicitly call *init\_printing* at the beginning to see results in a human-friendly output.

```
[ ]: import time
      from sympy import *
      # This will make all further
      # examples pretty print with unicode characters
      init_printing(use_unicode=True)
```

**DIFFERENCE:** In *SymPy*, one must explicitly declare which variables can be used in symbolic calculations.

```
[ ]: #help(symbols)
```

```
[ ]: x, y, z, t = symbols('x y z t')
```

```
[ ]: start_time = time.time()
```

Derivatives:

```
[ ]: #help(diff)
```

```
[ ]: diff(sin(x)*exp(x), x)
```

```
[ ]: 
$$e^x \sin(x) + e^x \cos(x)$$

```

```
[ ]: #help(factor)
```

```
[ ]: factor(diff(exp(x*y*z), x, y, y, z, z, z, z))
```

```
[ ]: 
$$x^3 y^2 (xyz + 4) (x^2 y^2 z^2 + 10xyz + 12) e^{xyz}$$

```

Integrals:

```
[ ]: #help(integrate)
```

```
[ ]: integrate(a, x)
```

```
[ ]:  $e^x \sin(x)$ 
```

```
[ ]: #help(powsimp)
```

```
[ ]: powsimp(integrate(sin(x**2), (x, -oo, oo)), force=true)
```

```
[ ]:  $\frac{\sqrt{2}\sqrt{\pi}}{2}$ 
```

```
[ ]: integrate(exp(-x**2 - y**2), (x, -oo, oo), (y, -oo, oo))
```

```
[ ]:  $\pi$ 
```

Limits:

```
[ ]: #help(limit)
```

```
[ ]: limit(sin(x)/x, x, 0)
```

```
[ ]: 1
```

Equations:

```
[ ]: #help(solve)
```

```
[ ]: solve(x**2 - 2, x)
```

```
[ ]:  $[-\sqrt{2}, \sqrt{2}]$ 
```

```
[ ]: # function definition
y = Function('y')
```

```
[ ]: #help(simplify)
```

```
[ ]: #help(dsolve)
```

```
[ ]: simplify(dsolve(Eq(y(t).diff(t, t) - y(t), exp(t)), y(t)))
```

```
[ ]:  $y(t) = \left(C_2 + \frac{(2C_1 + t)e^{2t}}{2}\right)e^{-t}$ 
```

Series expansion:

```
[ ]: #help(series)
```

```
[ ]: f = tan(x)
series(f, x, 2, 6, "+")
```

```
[ ]:
```

$$\begin{aligned} & \tan(2) + (1 + \tan^2(2))(x-2) + (x-2)^2(\tan^3(2) + \tan(2)) + (x-2)^3 \cdot \\ & \left(\frac{1}{3} + \frac{4\tan^2(2)}{3} + \tan^4(2)\right) + (x-2)^4 \left(\tan^5(2) + \frac{5\tan^3(2)}{3} + \frac{2\tan(2)}{3}\right) + (x-2)^5 \cdot \\ & \left(\frac{2}{15} + \frac{17\tan^2(2)}{15} + 2\tan^4(2) + \tan^6(2)\right) + O\left((x-2)^6; x \rightarrow 2\right) \end{aligned}$$

Total time for the section:

```
[ ]: print("--- %s seconds ---" % (time.time() - start_time))
--- 3.0264062881469727 seconds ---
```

## 2.0.2 Algebra

```
[ ]: #clear all variables
%reset -f
```

```
[ ]: import time
from sympy import *
# This will make all further
# examples pretty print with unicode characters
init_printing(use_unicode=True)
```

```
[ ]: a1, a2, a3, a4, b1, b2, b3, b4 = symbols('a1 a2 a3 a4 b1 b2 b3 b4')
```

```
[ ]: start_time = time.time()
```

```
[ ]: A = Matrix([[a1, a2], [a3, a4]])
      B = Matrix([[b1, b2], [b3, b4]])
```

Inverse:

```
[ ]: A.inv()
```

```
[ ]: 
$$\begin{bmatrix} \frac{a_4}{a_1 a_4 - a_2 a_3} & -\frac{a_2}{a_1 a_4 - a_2 a_3} \\ -\frac{a_3}{a_1 a_4 - a_2 a_3} & \frac{a_1}{a_1 a_4 - a_2 a_3} \end{bmatrix}$$

```

Matrix product:

```
[ ]: A*B
```

```
[ ]: 
$$\begin{bmatrix} a_1 b_1 + a_2 b_3 & a_1 b_2 + a_2 b_4 \\ a_3 b_1 + a_4 b_3 & a_3 b_2 + a_4 b_4 \end{bmatrix}$$

```

Transpose:

```
[ ]: #help(transpose)
```

```
[ ]: A.T
```

```
[ ]: 
$$\begin{bmatrix} a_1 & a_3 \\ a_2 & a_4 \end{bmatrix}$$

```

Determinant:

```
[ ]: #help(det)
```

```
[ ]: A.det()
```

```
[ ]: a1a4 - a2a3
```

Trace:

```
[ ]: #help(trace)
```

```
[ ]: A.trace()
```

```
[ ]: a1 + a4
```

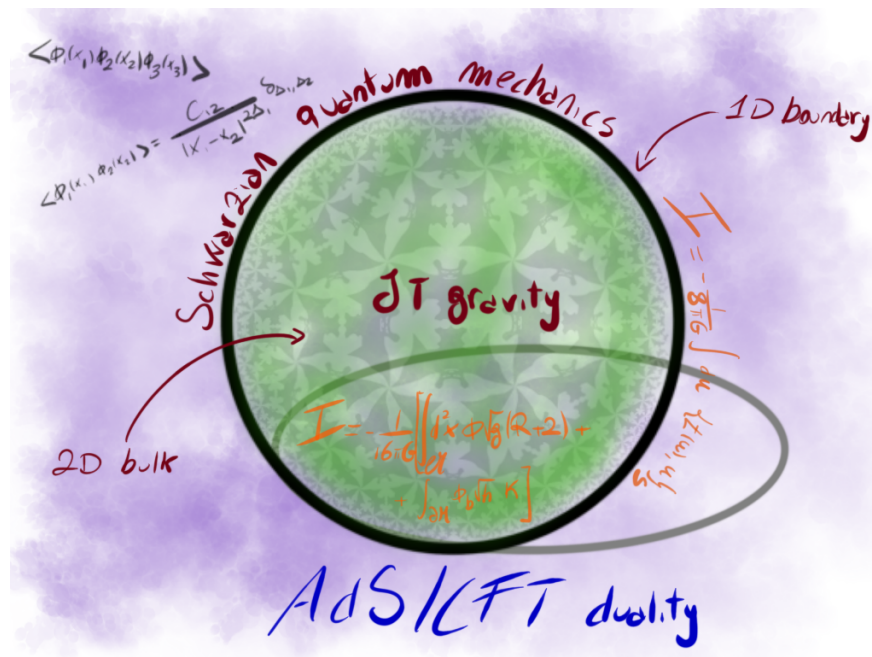
Time evaluation of the computations

```
[ ]: print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 0.4828970432281494 seconds ---
```

### 3 Packages for GR

We study the application of symbolic computation in General Relativity (GR) in low dimension (1+1D) following [Maldacena et al's article](#) for some computations, namely the first ones present in the article.



Jackiw-Teitelboim (JT) gravity is one of the basic theory of gravity you can have in 1+1D space-time which is physically consistent, where you couple a scalar field with gravity. It is possible to realize a duality theory between JT gravity and Schwarzian quantum mechanics (SQM).

### 3.1 Eisteinpy exploration

EinsteinPy is an open source, pure Python package dedicated to problems arising in General Relativity and gravitational physics, such as geodesics plotting for Schwarzschild, Kerr and Kerr-Newman space-time models, 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. 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 the [source code](#) of EinsteinPy.

EinsteinPy requires the following Python packages:

1. NumPy, for basic numerical routines
2. Astropy, for physical units and time handling
3. Matplotlib, for static geodesics plotting and visualizations.
4. Plotly, for interactive geodesics plotting and visualizations.
5. SciPy, for solving ordinary differential equations.
6. SymPy, for symbolic calculations related to GR.
7. Numba, for accelerating the code

Version used:

```
einsteinpy==0.4.0
```

```
[ ]: #clear all variables
%reset -f
```

```
[ ]: from sympy import *
import inspect
from einsteinpy.symbolic import *

init_printing(use_unicode=True)
```

Poincaré patch for  $AdS_2$  space:

```
[ ]: #help(MetricTensor)
```

**DIFFERENCE:** *EinsteinPy does not compute all the basic GR elements at once, so each time you have to call for an element you want to and then GraviPy compute it. In RGTC all the components are computed at once and stored, so the execution time is faster, but the memory storage capacity is easily fulfilled.*

```
[ ]: # define some symbolic variables
nt, nz = symbols('nt nz') # coordinates of the normal vector
# create a coordinate four-vector object instantiating
# the coordinate class
## Bulk space
syms = symbols('t z')
t,z = syms
## Boundary space
syms1 = symbols('u')
```

```

u = syms1
# define a matrix of a metric tensor components
Metric= diag(1/z**2, 1/z**2).tolist()
# create a metric tensor object instantiating the MetricTensor class
g = MetricTensor(Metric, syms)

```

### 3.1.1 Conventions

*Einsteinpy*:

1. The letters 'u' (=Up) and 'l' (=lower) to denote each free tensor index

*RGTC*:

1. The letters 'U' (=Up) and 'd' (=down) to denote each free tensor index
2. Finally, covariant differentiations can be denoted by preceding the differentiation index with the letter 'j' (resembling ';')
3. Contractions can be denoted by replacing the contracted indices by the same letter (Upper and lower case) while the symbol dollar can be used to separate different tensors

```
[ ]: g.tensor()
```

```
[ ]:  $\begin{bmatrix} \frac{1}{z^2} & 0 \\ 0 & \frac{1}{z^2} \end{bmatrix}$ 
```

### 3.1.2 Christoffel symbols

The Christoffel class represents Christoffel symbols. Components of the Christoffel objects are computed from the formula:

$$\Gamma_{\mu\nu}^{\sigma} = \frac{1}{2}g^{\sigma\rho}(g_{\rho\mu,\nu} + g_{\rho\nu,\mu} - g_{\mu\nu,\rho})$$

```
[ ]: chr = ChristoffelSymbols.from_metric(g)
chr.tensor()
```

```
[ ]:  $\begin{bmatrix} \begin{bmatrix} 0 & -\frac{1}{z} \\ -\frac{1}{z} & 0 \end{bmatrix} & \begin{bmatrix} \frac{1}{z} & 0 \\ 0 & -\frac{1}{z} \end{bmatrix} \end{bmatrix}$ 
```

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

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

### 3.1.3 Riemann Tensor

$$R_{\mu\beta\nu}^{\alpha} = \nabla_{\beta}\Gamma_{\mu\nu}^{\alpha} - \nabla_{\nu}\Gamma_{\beta\mu}^{\alpha} = \partial_{\beta}\Gamma_{\mu\nu}^{\alpha} - \partial_{\nu}\Gamma_{\beta\mu}^{\alpha} + \Gamma_{\mu\nu}^{\lambda}\Gamma_{\beta\lambda}^{\alpha} - \Gamma_{\beta\mu}^{\lambda}\Gamma_{\lambda\nu}^{\alpha}$$

```
[ ]: rm = RiemannCurvatureTensor.from_christoffels(chr)
rm.tensor()
```

```
[ ]:
```



$$\begin{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & -\frac{1}{z^2} \\ \frac{1}{z^2} & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & \frac{1}{z^2} \\ -\frac{1}{z^2} & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \end{bmatrix}$$

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

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

### 3.1.4 Ricci tensor

$$R_{\mu\nu} = R_{\mu\alpha\nu}^{\alpha}$$

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

```
[ ]:  $\begin{bmatrix} -\frac{1}{z^2} & 0 \\ 0 & -\frac{1}{z^2} \end{bmatrix}$ 
```

```
[ ]: Ric.config
```

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

### 3.1.5 Ricci scalar

$$R = R_{\mu}^{\mu}$$

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

```
[ ]: -2
```

### 3.1.6 Einstein tensor

$$G_{\nu\mu} = R_{\nu\mu} - \frac{1}{2}Rg_{\nu\mu}$$

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

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

```
[ ]: g.config
```

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

Evaluating tangent vector and normal vector:

```
[ ]: # Functions definition
T = Function('T')
Z = Function('Z')
```

```
[ ]: TU = GenericVector(Array([T(u).diff(u), Z(u).diff(u)]), syms, 'u', g)
TU.tensor()
```

```
[ ]:  $\left[ \frac{d}{du}T(u) \quad \frac{d}{du}Z(u) \right]$ 
```

```
[ ]: nU = GenericVector(Array([nt, nz]), syms, 'u', g)
nU.tensor()
```

```
[ ]:  $\begin{bmatrix} nt & nz \end{bmatrix}$ 
```

```
[ ]: nd = nU.change_config('1')
nd.tensor()
```

```
[ ]:  $\begin{bmatrix} \frac{nt}{z^2} & \frac{nz}{z^2} \end{bmatrix}$ 
```

Normalising the normal vector:

$$T^\mu n_\mu = 0$$

$$n^\mu n_\mu = 1$$

```
[ ]: nsolvedU = solve([Eq(simplify(tensorproduct(TU.tensor(),nd.tensor()).tomatrix().
    ↳trace()),0),
                      Eq(simplify(tensorproduct(nU.tensor(),nd.tensor()).tomatrix().
    ↳trace()),1)],(nt,nz))
nt, nz = nsolvedU[1]
nU = GenericVector(Array([nt, nz]), syms, 'u', g)
nd= nU.change_config('1')
```

```
[ ]: nU.tensor().subs(z, Z(u))
```

```
[ ]:  $\left[ \sqrt{\frac{1}{\left(\frac{d}{du}T(u)\right)^2 + \left(\frac{d}{du}Z(u)\right)^2}} Z(u) \frac{d}{du}Z(u) \quad - \sqrt{\frac{1}{\left(\frac{d}{du}T(u)\right)^2 + \left(\frac{d}{du}Z(u)\right)^2}} Z(u) \frac{d}{du}T(u) \right]$ 
```

```
[ ]: nd.tensor().subs(z, Z(u))
```

```
[ ]:  $\left[ \frac{\sqrt{\frac{1}{\left(\frac{d}{du}T(u)\right)^2 + \left(\frac{d}{du}Z(u)\right)^2}} \frac{d}{du}Z(u)}{Z(u)} \quad - \frac{\sqrt{\frac{1}{\left(\frac{d}{du}T(u)\right)^2 + \left(\frac{d}{du}Z(u)\right)^2}} \frac{d}{du}T(u)}{Z(u)} \right]$ 
```

## 4 GraviPy exploration

GraviPy is a Tensor Calculus Package for General Relativity based on SymPy.

GraviPy depends on:

1. Python (version  $\geq 3.6$ )
2. SymPy (version  $\geq 1.4$ )

### 3. JupyterLab (version $\geq 1.1.3$ , optional)

JupyterLab environment isn't required but it makes it easier to work with complex mathematical expressions.

To install GraviPy, simply run

```
$ pip install gravipy
```

Version used:

```
GraviPy==0.2.0
```

```
[ ]: #clear all variables
      %reset -f
```

```
[ ]: from gravipy.tensorial import *
      from sympy import *
      import inspect
      init_printing()
      from IPython.display import display, Math
      from sympy import latex
```

Poincaré patch for  $AdS_2$  space:

```
[ ]: #help(MetricTensor)
```

**DIFFERENCE:** *GraviPy does not compute all the basic GR elements at once, so each time you have to call for an element you want to and then GraviPy compute it. In RGTC all the components are computed at once and stored, so the execution time is faster, but the memory storage capacity is easily fulfilled.*

```
[ ]: #dimension of the space-time
d=2
# define some symbolic variables
alpha, gamma, delta = symbols('alpha gamma delta')
nt, nz = symbols('nt nz')
u = symbols('u')
t, z = symbols('t z')
# create a coordinate four-vector object instantiating
# the coordinate class
## Bulk space
x = Coordinates('\chi', [t, z])
## Boundary space
y = Coordinates('\chi1', [u])
# define a matrix of a metric tensor components
Metric = diag(1/z**2, 1/z**2)
# create a metric tensor object instantiating the MetricTensor class
g = MetricTensor('g', x, Metric)
```

### 4.0.1 Conventions

1bis. Gravipy: Each component of any tensor object, can be computed by calling the appropriate instance of the GeneralTensor subclass with indices as arguments. The covariant indices take positive integer values (1, 2, ..., dim). The contravariant indices take negative values (-dim, ..., -2, -1).

RGTC:

1. The letters 'U' (=Up) and 'd' (=down) to denote each free tensor index
2. Finally, covariant differentiations can be denoted by preceding the differentiation index with the letter 'j' (resembling ';')
3. Contractions can be denoted by replacing the contracted indices by the same letter (Upper and lower case) while the symbol dollar can be used to separate different tensors

```
[ ]: g(A11, A11)
```

```
[ ]:  $\begin{bmatrix} \frac{1}{z^2} & 0 \\ 0 & \frac{1}{z^2} \end{bmatrix}$ 
```

```
[ ]: g(-A11, -A11)
```

```
[ ]:  $\begin{bmatrix} z^2 & 0 \\ 0 & z^2 \end{bmatrix}$ 
```

### 4.0.2 Christoffel symbols

The Christoffel class represents Christoffel symbols. Components of the Christoffel objects are computed from the formula:

$$\Gamma_{\mu\nu}^{\sigma} = \frac{1}{2}g^{\sigma\rho}(g_{\rho\mu,\nu} + g_{\rho\nu,\mu} - g_{\mu\nu,\rho})$$

```
[ ]: chr = Christoffel('chr', g)
chr(-A11, A11, A11)
```

```
[ ]:  $\begin{bmatrix} 0 & -\frac{1}{z} \\ -\frac{1}{z} & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{z} & 0 \\ 0 & -\frac{1}{z} \end{bmatrix}$ 
```

### 4.0.3 Riemann Tensor

$$R_{\mu\beta\nu}^{\alpha} = \nabla_{\beta}\Gamma_{\mu\nu}^{\alpha} - \nabla_{\nu}\Gamma_{\beta\mu}^{\alpha} = \partial_{\beta}\Gamma_{\mu\nu}^{\alpha} - \partial_{\nu}\Gamma_{\beta\mu}^{\alpha} + \Gamma_{\mu\nu}^{\lambda}\Gamma_{\beta\lambda}^{\alpha} - \Gamma_{\beta\mu}^{\lambda}\Gamma_{\lambda\nu}^{\alpha}$$

```
[ ]: Rm = Riemann('Rm', g)
Rm(-A11, A11, A11, A11)
```

```
[ ]:  $\begin{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & -\frac{1}{z^2} \\ \frac{1}{z^2} & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & \frac{1}{z^2} \\ -\frac{1}{z^2} & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \end{bmatrix}$ 
```

#### 4.0.4 Ricci tensor

$$R_{\mu\nu} = R_{\mu\alpha\nu}^{\alpha}$$

```
[ ]: Ri = Ricci('Ri', g)
      Ri(All, All)
```

```
[ ]:  $\begin{bmatrix} -\frac{1}{z^2} & 0 \\ 0 & -\frac{1}{z^2} \end{bmatrix}$ 
```

#### 4.0.5 Ricci scalar

$$R = R_{\mu}^{\mu}$$

```
[ ]: R = Ri.scalar()
      R
```

```
[ ]: -2
```

#### 4.0.6 Einstein tensor

$$G_{\nu\mu} = R_{\nu\mu} - \frac{1}{2}Rg_{\nu\mu}$$

```
[ ]: G = Einstein('G', Ri)
      G(All, All)
```

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

Evaluating tangent vector and normal vector:

```
[ ]: # functions definition
      T = Function('T')
      Z = Function('Z')
```

**DIFFERENCE:** *One must define tensors associated to a metric in order to compute covariant and controvariant components*

```
[ ]: # Tensor always define the covariant (lower) components
      Tangent = Tensor('V', 1, g, components=Matrix([T(u).diff(u)/z**2, Z(u).diff(u)/
      ↪ z**2]))
      Tangent(-All)
```

```
[ ]:  $\begin{bmatrix} \frac{d}{du}T(u) & \frac{d}{du}Z(u) \end{bmatrix}$ 
```

```
[ ]: #Tensor always define the covariant (lower) components
      Normal = Tensor('n', 1, g, components=Matrix([nt/z**2, nz/z**2]))
      Normal(-All)
```

```
[ ]:  $\begin{bmatrix} nt & nz \end{bmatrix}$ 
```

Normalising the normal vector:

$$T^\mu n_\mu = 0$$

$$n^\mu n_\mu = 1$$

```
[ ]: nsolvedU = solve([Eq(Tangent(-All).dot(Normal(All)), 0),
                        Eq(Normal(-All).dot(Normal(All)), 1)],(nt, nz))
nt, nz = nsolvedU[1]
nt = nt.subs({Z(u):z}).subs(T(u), t)
nz = nz.subs({Z(u):z}).subs(T(u), t)
```

Induced metric:

**DIFFERENCE:** in *GraviPy*, there is no simple way to perform tensorial computations in a manner that properly evaluate covariant and contravariant terms.

There is some function implemented in *Sympy*, but still not as simple as the one in *RGTC*:  
<https://docs.sympy.org/latest/modules/tensor/tensor.html>

TensMul( )

```
[ ]: InducedMetric = trace(g(All, All).subs(z, Z(u))*(Tangent(-All).T*Tangent(-All)))
h = MetricTensor('h', y, Matrix([InducedMetric]))
h(All, All)
```

```
[ ]: [ (d/dt T(u))^2 / Z^2(u) + (d/dt Z(u))^2 / Z^2(u) ]
```

```
[ ]: Normal = Tensor('n', 1, g, components=Matrix([nt/z**2, nz/z**2]))
Normal(-All)
```

```
[ ]: [ z * sqrt(1 / ((d/dt t)^2 + (d/dt z)^2)) * d/dt z, -z * sqrt(1 / ((d/dt t)^2 + (d/dt z)^2)) * d/dt t ]
```

Stress-energy tensor:

$$\tilde{T}_{\mu\nu} = 8\pi G T_{\mu\nu} = (\nabla_\mu \nabla_\nu - g_{\mu\nu} \nabla^2 + g_{\mu\nu}) \phi$$

```
[ ]: #scalar dilaton field
phi = Tensor('phi', 0, g)
phi()
```

```
[ ]: phi(t, z)
```

Covariant derivative:

**DIFFERENCE:** One must compute each and every covariant derivate of interest and then create a matrix to store them since `>Tensor()`

needs a *Sympy* Matrix

```
[ ]: for k in range(1,d+1):
      for j in range(1,d+1):
          phi.covariantD(k,j).simplify()
```

```
for k in phi.covariant_derivative_components:
    display(Math(str(k) + ':' +
                  + latex(phi.covariant_derivative_components[k])))
```

$$(1,) : \frac{\partial}{\partial t} \phi(t, z)$$

$$(2,) : \frac{\partial}{\partial z} \phi(t, z)$$

$$(1,1) : \frac{z \frac{\partial^2}{\partial t^2} \phi(t, z) - \frac{\partial}{\partial z} \phi(t, z)}{z}$$

$$(1,2) : \frac{z \frac{\partial^2}{\partial z \partial t} \phi(t, z) + \frac{\partial}{\partial t} \phi(t, z)}{z}$$

$$(2,1) : \frac{z \frac{\partial^2}{\partial z \partial t} \phi(t, z) + \frac{\partial}{\partial t} \phi(t, z)}{z}$$

$$(2,2) : \frac{z \frac{\partial^2}{\partial z^2} \phi(t, z) + \frac{\partial}{\partial z} \phi(t, z)}{z}$$

```
[ ]: #initializing matrix
stressenergy = zeros(2)
for k in range(0,d):
    for j in range(0,d):
        stressenergy[k, j] = phi.covariant_derivative_components[(k+1, j+1)]
stressenergy += g(All, All) * phi() - g(All, All) * trace(g(-All, -All) *
↪ stressenergy)
stressenergy = simplify(stressenergy)
Tphi=Tensor('Tphi', 2, g, components=Matrix(stressenergy))
Tphi(All, All)
```

```
[ ]:
```

$$\begin{bmatrix} -\frac{\partial^2}{\partial z^2} \phi(t, z) - \frac{\frac{\partial}{\partial z} \phi(t, z)}{z} + \frac{\phi(t, z)}{z^2} & \frac{\partial^2}{\partial z \partial t} \phi(t, z) + \frac{\frac{\partial}{\partial t} \phi(t, z)}{z} \\ \frac{\partial^2}{\partial z \partial t} \phi(t, z) + \frac{\frac{\partial}{\partial t} \phi(t, z)}{z} & -\frac{\partial^2}{\partial t^2} \phi(t, z) + \frac{\frac{\partial}{\partial z} \phi(t, z)}{z} + \frac{\phi(t, z)}{z^2} \end{bmatrix}$$

Checking Maldacena's et al. solution

```
[ ]: # functions definition
phi_0 = (alpha + gamma*t + delta*(t**2 + z**2))/z
```

```
[ ]: cancel(simplify(stressenergy.subs(phi(), phi_0)))
```

```
[ ]:
```

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

## 5 FROM SYMBOLIC TO NUMERICAL EVALUATION

It is possible to convert symbolic computation in numerical computation using Sympy:

1. lambdify
2. subs/evalf

3. lambdify-numpy
4. ufuncify
5. lambdify-cupy
6. Aesara

Example with

`lambdify( )`

```
[83]: #clear all variables
      %reset -f
```

```
[84]: from sympy import *
      from sympy.abc import x #symbol imported
      expr = sin(x)/x
      f = lambdify(x, expr)
      f(3.14)
```

```
[84]: 0.00050721430461364
```

Comparison between different symbolic-to-numeric computation

Tool	Speed	Qualities	Dependencies
subs/evalf	50us	Simple	None
lambdify	1us	Scalar functions	math
lambdify-numpy	10ns	Vector functions	numpy
ufuncify	10ns	Complex vector expressions	f2py,Cython
lambdify-cupy	10ns	Vector functions on GPUs	cupy
Aesara	10ns	Many outputs,CSE,GPU's	Aesara

For the GR part EinsteinPy implements both numeric and symbolic computation between SymPy and NumPy

A rather faster approach to convert symbolic computation in numerical computation in Mathematica is done by adding N in front of many symbolic-valued functions.

For instance

`NSolve[ ]`

see [NSolve](#) for references.

## 6 Conclusion

1. Open source symbolic computation libraries in python:
  - **SymPy** is a pure Python library,that does computer algebra.
    - **SageMath** tries to gather together all the major open source mathematics software, and glue it together into a useful system. In fact, Sage includes SymPy as one of its systems;



- **Cadabra** is a symbolic computer algebra system (CAS) designed specifically for the solution of problems encountered in field theory. It has extensive functionality for tensor computer algebra, tensor polynomial simplification including multi-term symmetries, fermions and anti-commuting variables, Clifford algebras and Fierz transformations, component computations, implicit coordinate dependence, multiple index types and many more. The input format is a subset of TeX. Cadabra is a tensor algebra and (quantum) field theory system using SymPy for scalar algebra.
- 2. SageMath, Cadabra, SymPy, EinsteinPy and GraviPy can be used in [Jupyter Notebooks](#)

Here are some comments on our explorations:

1. For basic symbolic calculations, we found no major differences between Mathematica and SymPy. A few points on which SymPy may be improved are the following:
  - In SymPy, one must to explicitly call a function to get a nice visualization of the results. In Mathematica, does is is done automatically and it also gives the opportunity to visualize the corresponding code by pressing CTRL+↑+N
  - Because of the nature of variables in Python, in SymPy one must explicitly declare what variables are symbols that can be used for symbolic computation. In Mathematica variables are symbolic by default. This may seem of no concern, but when have to work with an large set of variables, initializing them is tedious.
  - Mathematic is appreciable (although not dramatically) faster. These were the timings during our tests on my personal computer

Program	Analysis part	Algebra Part
Mathematica	1.016 s	0.094 s
SymPy	2.545 s	0.274 s

2. For GR calculations, we found some major differences between RGTC and GraviPy and/or EinsteinPy:
  - Both **GraviPy** and **EinsteinPy** do not compute all the basic GR elements at once, i.e. Christoffel symbols, Reimamann tensor, Ricci tensor and scalar and the Weyl tensor.
  - **RGTC** reports if the space is comformally flat and what type of space-time is just by giving the metric tensor.
  - In **EinsteinPy**, there is no function implemennting covariant derivatives.
  - Neither **GraviPy** nor **EinsteinPy** provide a simple way to perform tensorial computations that properly evaluate covariant and controvariant terms.
  - **GraviPy** and **EinsteinPy** are not as user-friendly as **RGTC** when it comes to create covariant derivatives along an arbitrary vector field. In fact, there are still several technical problems to be solved to achieve that ( [ISSUES ON GITHUB](#) ).
  - It was not possible with either **GraviPy** or **EinsteinPy** to compute in an easy way all the results reproduce in the Mathematica test notebook, perhaps some solutions are available but not of straightforward implementation.

Overall, Sympy is a valid python open source package to perform symbolic computations. It covers most use cases in calculus and algebra and can be a powerful computational tool.

Concerning symbolic calculations in GR, GraviPy and EinsteinPy are, at present, not as advanced and flexible as RGTC. However, they are open source, community-driven projects: over time,

updates and new features will be added to the code, and we hope them to improve in the next versions.

**NOTE:**

This notebook was realised as a project for a Master course (“Abilità informatiche e telematiche”) in theoretical physics at the University of Trieste.