PyOGRe

April 27, 2022

1 PyOGRe: A Python Object-Oriented General Relativity Package

Documentation for v1.0.0 (January 2022)

By **Barak Shoshany** (website) and **Jared Wogan** (website)

GitHub repository

PyPi Project

2 Introduction

2.1 Summary

PyOGRe is the Python port of the Mathematica package OGRe, written by Professor Barak Shoshany at Brock University. OGRe is a tensor calculus package that is designed to be both powerful and user-friendly. OGRe can perform calculations involving tensors extremely quickly, which could often take hours to do by hand. It is extremely easy to pick up and use, with easy to learn syntax. Naturally, the package has applications in general relativity where tensors are used abundantly and was the focus point during the development of the package. There is no restriction preventing the package from being used for other applications, like electromagnetism, quantum mechanics, or other fields.

Tensors are abstract mathematical objects that can be represented by multidimensional arrays. A tensor may be represented by anything from a scalar, a vector, a matrix, to a N-dimensional array. Each array is a specific representation of a tensor, but this representation is not the only valid representation. By performing a change of coordinates, or changing the index configuration, we can produce another completely valid representation of the same tensor. It is thus crucial that we specify both a set of coordinates and an index configuration when calling a specific array a representation of a tensor. Take, for example, the (rank 2) tensor $T_{\mu\nu}$ below:

$$T_{\mu\nu}(t,x,y,z) = \begin{pmatrix} T_{tt} & T_{tx} & T_{ty} & T_{tz} \\ T_{xt} & T_{xx} & T_{xy} & T_{xz} \\ T_{yt} & T_{yx} & T_{yy} & T_{yz} \\ T_{zt} & T_{zx} & T_{zy} & T_{zz} \end{pmatrix} = \begin{pmatrix} T_{00} & T_{01} & T_{02} & T_{03} \\ T_{10} & T_{11} & T_{12} & T_{13} \\ T_{20} & T_{21} & T_{22} & T_{23} \\ T_{30} & T_{31} & T_{32} & T_{33} \end{pmatrix}$$

Above is a single representation of the tensor T, with two lower indices, written in standard Cartesian coordinates. We call T a rank two tensor because it requires two indices to specify a component of the tensor; this means a scalar can represent a rank zero tensor, a vector can represent a rank

one tensor, and so on (note how we do not say a scalar is a tensor, but instead that it can represent a tensor). If we wrote the same tensor in a different coordinate system, or with a different index configuration, the components would be completely different. Since we are working with representations of abstract objects, it is clear that we must be careful and ensure that all representations of a given tensor truly do represent the same tensor.

When working with tensors, it is natural to want to combine them in different ways. PyOGRe allows the user to perform calculations involving tensors in a straightforward manner. The user will only need to input a tensor once with a specified choice of coordinates and an index configuration, then behind the scenes, PyOGRe will automatically ensure the correct representation of the tensor is used. This makes the often complicated and/or confusing expressions involving tensors straightforward, taking away the unnecessary complexities of the calculation. Some of the possible operations that PyOGRe can perform are:

- Addition
- Subtraction
- Multiplication by a scalar
- Trace
- Contraction
- Partial and Covariant Derivatives

The documentation provided below will ensure that a proper theoretical understanding of the mathematics behind the operations is achieved, then, we will explain how each calculation can be performed using PyOGRe.

Behind the scenes, PyOGRe takes an object-oriented design and stores each tensor as a single object (we will explain object-oriented design in a later section). Each object stores all the representations of the tensor, where each representation is stored as an array. We ensure that all representations stored within the tensor object truly represents the same tensor by preventing the user from modifying the data once it has been created. Preventing the user from changing the data of a tensor is a fundamental part of the object-oriented design, as it allows a series of assumptions to be made about the tensor, greatly improving performance and preventing errors.

Instead of allowing the user to modify the data of a tensor, PyOGRe instead allows the user to request specific reprentations of any tensor. Handling the data of a tensor in this manner allows PyOGRe to be much more efficient; if a specific representation already exists, it will not be recalculated, while still giving the user the ability to request a tensor in different representations. Secondly, if a represntation of a tensor is calculated in some intermediate step of a calculation, all intermediate representations are stored as well, which can provide another increase in efficiency.

The remainder of this document will outline why we have decided to port OGRe to Python, a brief list of the features, the meaning of object-oriented design, the differences between the original OGRe package and PyOGRe, and finally documentation of each module of the package.

2.2 Motivation for the Python Port

Porting a piece of software is simply the act of converting a piece of software from one programming language to another. A port is meant to achieve the same level of functionality. PyOGRe is simply a port of the OGRe package written in Mathematica to Python. Both versions of the package will retain near identical syntax, as well as cross compatibility, so that work in one package may be freely moved to the other.

The decision to port the package from Mathematica to Python consists of several motivating factors. One of the primary reasons is because Mathematica requires a license which must be purchased. This makes accessing the package difficult for people who may not have the required funding for a Mathematica license, which is where Python can help. Python is **freely** available for anyone to install, so by porting the package to Python, we are removing the paywall that the Mathematica version currently imposes.

Additionally, while Mathematica is extremely powerful and popular within the mathematics or physics communities, Python is far more popular in general. Python has been seeing a surge in popularity over the past years and is now becoming one of the most common languages beginners start with, especially within physics. Porting the package to Python will thus vastly improve the number of potential users, as well as the **accessibility** over the existing package in Mathematica.

Finally, in order to port the package to another language, we needed to consider which languages had support for **symbolic computation**. Again, Python is a clear option over a language such as C/C++, as it is both popular for scientific computation already, and has support for symbolic computation through a package called SymPy. We have used SymPy extensively when developing PyOGRe and have found it to be both easy to use and powerful. SymPy is very easy to pick up and learn, and there is very little the end user will have to know about SymPy in order to use PyOGRe. Of course, since Python is also an object-oriented language, we were able to stay true to the object-oriented design philosophy, which was not entirely possible with Mathematica.

PyOGRe is not meant to be a replacement for OGRe in any way, but merely a complimentary package to the existing package, giving the user flexibility to perform computation in either Python or Mathematica. All of the same features found in the original package are available in PyOGRe or will be made available in the future. We have also made it possible for the user to export and import tensors created in one version of the package to the other version.

2.3 Features

PyOGRe is currently under active development and we are constantly adding new functionality. The following is a brief outline of the main features that can currently be found in PyOGRe:

- Define coordinate systems and any transformation rules between them. Tensor components in any representation can then be transformed automatically as needed.
- Define metrics which can then be used to define any arbitrary tensor. The metric associated with a tensor will be used to raise and lower indices as needed.
- Display any tensor object in any coordinate system as an array, or as a list of the unique non-zero components. Metrics can additionally be displayed as a line element.
  - When displaying a tensor, substitutions can be made for any variable or function present. Additionally, a function may be specified that will be mapped to each component of the tensor.
- Export tensors to a file so that they can later be imported into a new session or into the Mathematica version.
- A simple API for performing calculations on tensors, including addition, subtraction, multiplication by a scalar, trace, contraction, as well as partial and covariant derivatives.
- Built-in tensors for commonly used coordinate systems and metrics.
- Built-in functions for calculating the Christoffel symbols (the Levi-Cevita connection), Riemann tensor, Ricci tensor, Ricci scalar, Einstein Tensor, curve Lagrangian, and volume element from a metric, as well as the norm squared of any tensor.

- Calculate the geodesic equations in terms of a user defined curve parameter (affine parameter), using either the Christoffel symbols or the curve Lagrangian (for spacetime metrics, the geodesic equations can be calculated in terms of the time coordinate).
- Designed to be performant using optimized algorithms for common operations (these functions can be used on any SymPy array).
- Quick and easy to install straight from PyPI (supports Python 3.6 and above, previous versions untested).
- Command line and Jupyter notebook support.
- Clear and well documented source code, complete with examples.
- Open source and available for all to use.
- Easily extendible and modifiable.
- Under active development and will be updated regularly.

2.4 Object-Oriented Design Philosophy

Object-oriented programming (OOP) simply refers to a programming paradigm in which objects are used to combine data and functionality. Each object contains some amount of data (often called **attributes**), as well as functions that can operate on that data (often called **methods**). Each **object** belongs to a **class** that the user defines and can be thought of as a blueprint. The class tells the end user what kind of data each object will have, and how that data should be stored. The class also tells the end user what kind of functions each object will have, and how those functions should be implemented.

An extremely important aspect of object-oriented design is the idea of **encapsulation**. Encapsulation is the process of hiding the implementation details of an object, and only exposing the interface to the user. What this means is that the user may only access the data of an object through methods defined by the class but is unable to change or modify the actual data of the object directly. This enables a new idea, **class invariants**, which are assumptions about the object that should always be preserved. The invariance of a class allows the assumption that all data stored in each object remains valid, leading to more efficient and simpler code.

As we discussed previously, tensors are abstract mathematical objects, which makes object-oriented programming a natural choice. Each tensor is stored as a single, self-contained object. The tensor object stores all the required data about each tensor, including the components for each known or calculated representation. Since all tensors in PyOGRe share a common class, we only need to define operations on tensor objects once in an abstract manner, and PyOGRe will automatically be able to apply these operations to any tensor object.

The most important class invariant of a tensor object in PyOGRe are naturally the components of the tensor in each representation. Since each tensor stores these components in the object representing the tensor, it is crucial that each representation does indeed represent the tensor. This is done through encapsulation; the user is **not** allowed to access these components of the tensor once it has been defined. The components may only be modified by **private methods** (methods the user does not have access to) that preserve the invariance of the class. This prevents the user from accidentally violating the class invariance, and thus, the invariance of the tensor object. This again, allows us to work under the assumption that all the data stored inside a tensor object is valid, and that the data does in fact represent the tensor.

In PyOGRe, the user will initially define (or **construct**) a tensor in some specific representation once but will then never have to worry about coordinates or indices anymore. In fact, the user will

not even need to remember which coordinates or indices were used to construct the tensor in the first place. The user will simply be able to request the tensor in any representation they desire, or when performing calculations, PyOGRe will determine which representation is required in each context.

An important distinguishing feature between the Mathematica version (OGRe) and the Python version (PyOGRe) of the package is that in Python, we can truly define tensors as objects. In Mathematica, there is no notion of a class; all tensors are simply represented as an association. At heart, the Mathematica version operates as though it is object-oriented, and ensures that the "objects" are invariant, however the objects themselves are merely a list. This is because Mathematica is not an object-oriented programming language; in Mathematica, it is not possible to define class methods, as there are no classes to begin with. To ensure PyOGRe feels familiar to the user, we have also included functions alongside the class methods that maintain the same functionality.

2.5 Syntax Differences

To illustrate some differences between the two versions, and how Python allows us to adhere to the object-oriented paradigm more closely, consider the following. Suppose a user wanted to define the Minkowski metric. In OGRe, assuming the Cartesian coordinates were already defined, the user could define a new metric as follows:

```
TNewMetric["Minkowski", "Cartesian", DiagonalMatrix[{-1, 1, 1, 1}]]
```

while in Python, the user could instead do:

```
cartesian.new_metric("Minkowski", sympy.diagonal(-1, 1, 1, 1))
```

The difference is that in Python, defining a metric can be done by calling a method on a coordinate system. PyOGRe will then know automatically that the new metric is defined in terms of the Cartesian coordinates and will not have to be told explicitly. Of course, one can also create a new tensor by calling a method on the metric. If instead the user wanted to calculate the norm squared of a tensor, the user could use the method `calc_norm_squared` on the tensor in PyOGRe, where in OGRe one would instead have to write `TCalcNormSquared`, with the tensor as an argument. It should be noted though that the functional equivalents of these methods are also available in PyOGRe.

Another minute difference between the two versions, is that in OGRe, function names use camel case (e.g., TCalcNormSquared). In PyOGRe, we use snake case to adhere to common Python conventions (e.g., calc_norm_squared). This does not in any way change the functionality of the functions or methods, it is merely a slightly different naming scheme.

3 Installing and Loading the Package

3.1 Installing

Installing PyOGRe is done using PIP. Simply open a terminal, and type:

```
pip install PyOGRe
```

PIP will automatically find the latest version and install it to the default interpreter, including all dependencies. If you would like to update a current installation of PyOGRe, you may instead run:

```
pip install --upgrade PyOGRe
```

If you are looking for a specific version of PyOGRe, you may specify the version using the following command:

```
pip install PyOGRe==x.y.z
```

where **x.y.z** is the targeted version.

3.2 Importing PyOGRe

Importing PyOGRe is done in the same manner as any other Python package. We recommend also importing `SymPy`, which is used extensively by the package, and the `display` function from `IPython.display` if you are working within a Jupyter Notebook.

NOTE: Although tempting, never write something such as:

```
from PyOGRe import *
```

This is known as a wildcard import and can clutter the namespace, therefore we recommend importing PyOGRe as we have done below.

[ ]:
```
%load_ext autoreload
%autoreload 2
```

[ ]:
```
import sympy as sym
from IPython.display import display

import PyOGRe as og
```

If you are working in a terminal, call the `command_line_support()` function to tell PyOGRe that it should display results using ASCII characters. The default behaviour assumes you are working in a Jupyter Notebook (this can be set manually using `jupyter_support()`), and all outputs are rendered using Markdown and LaTeX.

[ ]:
```
og.command_line_support()
og.jupyter_support()
```

If you would like to see the current options the package is using, call the `get_options()` function.

[ ]:
```
og.get_options()
```

```
LATEX: True
ALL_SYMBOLS: (mu, nu, rho, sigma, kappa, lambda, alpha, beta, gamma, delta,
epsilon, zeta, theta, iota, xi, pi, tau, phi, chi, psi, omega)
CURVE_PARAMETER: lambda
INFO_ORDER: ('Name', 'Symbol', 'Type', 'Rank', 'Metric', 'Default Coordinates',
'Default Indices', 'Default Coordinates For', 'Coordinate Transformations',
'Indices Symbols', 'Tensors Using This Metric')
LIST_PER_LINE: 6
FONT_SIZE: 14
PARALLEL: False
```

One may also use the `doc()` function to find the documentation for the package, which is this notebook (also available in PDF format).

[ ]:
```
og.doc()
```

```
PyOGRe Documentation

Version: 1.0.0

PyOGRe is an Object-Oriented General Relativity Package for Python.
The full documentation is available at:
```

4 Creating and Displaying Tensor Objects

4.1 A Quick Overview of SymPy

Before we start working with PyOGRe and tensors, we will first introduce the necessary SymPy objects that are frequently used. There are three main SymPy commands that we should be aware of in order to make full use of PyOGRe:

- `symbols()`

  This function accepts a string such as "a b c", and will return a tuple of SymPy symbols (each set of characters separated by a space are converted to symbols, see the SymPy documentation for more information). SymPy symbols represent symbolic variables which can be used in mathematical expressions. It is important to note that symbols are treated as constants, so if the variable we would like to create is dependent on another variable, it should instead be created using a function.

- `Array()`

  The array function turns a list (or nested lists) into a SymPy array. SymPy arrays along with SymPy symbols are the most common building blocks used in PyOGRe.

- `Function()()`

  The SymPy Function class creates a function object. Functions are exactly like symbols, except they are not treated as constants.

[ ]:
```
t, x, y, z = sym.symbols('t x y z')
r, theta, phi = sym.symbols('r theta phi')

f = sym.Function("f")(t)

cartesian_symbols = sym.Array([t, x, y, z])
spherical_symbols = sym.Array([t, r, theta, phi])

display(f)
display(cartesian_symbols)
display(spherical_symbols)
```

$f(t)$

$$\begin{bmatrix} t & x & y & z \end{bmatrix}$$

$$\begin{bmatrix} t & r & \theta & \phi \end{bmatrix}$$

Above, we have created the symbols t, x, y, z, r, θ and ϕ which are used throughout the documentation. We then created a new function $f(t)$ using the symbol t we defined earlier. Finally, two SymPy arrays are created, one of which we will use to define Cartesian coordinates, and the second to define Spherical coordinates.

4.2 Defining Coordinate Systems

Now, before we can define tensors, we must first define the coordinate systems that we will be using. This documentation will be focused on general relativity, so we will be using 4-dimensional spacetime coordinate systems, however the package works equally well for other applications that may be in any number of dimensions.

In order to define a coordinate system in PyOGRe, we can use the `new_coordinates()` function:

[ ]:
```
print(og.new_coordinates.__doc__)
```

```
    (function) new_coordinates(name, components, indices, symbol,
transformations)

    Creates a new tensor object representing a coordinate system.

    `name`: Defines name of the tensor is used when displaying the coordinates.
    `components`: An array of the coordinate symbols.
    `indices`: Defines the indices of the coordinates, must be (1,).
    `symbol`: Defines the symbol used to represent the coordinates.
    `transformations`: Defines the coordinate transformations, but can be left
as none and added later.

    Example:

    >>> import sympy as sym
    >>> from PyOGRe import new_coordinates
    >>> new_coordinates("Cartesian", sym.Array([sym.Symbol("x"),
sym.Symbol("y"), sym.Symbol("z")]))

    This will create a coordinates object named Cartesian with coordinate
symbols x, y, and z.
```

At the bare minimum, we must supply both a name for the Coordinates object, along with the components. The **name** of the object will be displayed whenever we print the tensor (the name will also be the ID of the object when exported to Mathematica). The second argument being the **components** is a list of the coordinates themselves. The order of the coordinates matters as this order will be used when displaying components of other tensors that are defined using this coordinate system.

[ ]:
```
cartesian = og.new_coordinates(
    name="Cartesian",
    components=cartesian_symbols
)

spherical = og.new_coordinates(
    name="Spherical",
    components=spherical_symbols
)
```

4.3 Displaying PyOGRe Objects

4.3.1 Setting the Font Size

Before displaying our tensors, we may first want to choose a font size for the output. By default, PyOGRe uses a font size of 14, however this can be changed using the `set_font_size()` function. Passing an argument of `None` will print the current value used for the font size.

[ ]:
```
og.set_font_size()
```

```
14
```

4.3.2 Displaying as an Array

Let us now introduce how we can display our tensors. The first method that can be used in PyOGRe is the `show()` method. Using the `show()` method will display the tensor as an array, along with the tensor name and tensor symbol.

[ ]:
```
cartesian.show()
```

Cartesian:

$$x^{\mu} = \begin{pmatrix} t \\ x \\ y \\ z \end{pmatrix}$$

For any object that is not a Coordinates object, the `show()` method can also accept parameters. The first argument **coords** will be the coordinate system to use when displaying the object. The second argument **indices** can be used to specify an index configuration for the tensor. The argument **replace** can be supplied with the arguments to use in the SymPy `subs()` method. Additionally, the arguments **function** and **args** can be used to specify a function and the arguments for the function, that will be mapped to each element of the array representing the tensor. We will see plenty of examples as we go through the documentation.

4.3.3 Displaying as a List

The second method for displaying tensors is the `list()` method. With this method, we display the non-zero elements of the components of the tensor, along with the tensor name and symbol. If

there are no non-zero elements, the method will simply inform us there are no non-zero elements instead.

[ ]:
```
spherical.list()
```

Spherical:

$$x^t = t$$

$$x^r = r$$

$$x^\theta = \theta$$

$$x^\phi = \phi$$

Just like for the `show()` method, if the object we are calling the `list()` method on is not a Coordinates object, `list()` will accept the same arguments as detailed previously.

Additionally, if we would like to change the number of components that are displayed on each line, we may use the `set_list_per_line()` function. By default, PyOGRe will display up to 6 components per line. Just like when we set the font size, passing an argument of `None` will print the current value.

[ ]:
```
og.set_list_per_line()
```

```
6
```

4.3.4 Retrieving Information about a Tensor

Finally, we have the `info()` method, which will display more detailed information about the tensor. This includes the name, symbol, the coordinates used to define the tensor, the rank, the default indices, and any other tensors that are defined using the tensor the method was called on.

[ ]:
```
cartesian.info()
```

Cartesian:

Name: Cartesian

Symbol: x

Type: Coordinates

Rank: 1

Default Indices: (1,)

Default Coordinates For:

4.3.5 Getting the Rank of a Tensor

It may be useful to be able to request the rank of a tensor, which can be done by calling the `rank()` method. This method will simply return the rank of the tensor.

[ ]:
```
print(f"{cartesian.rank() = }")
print(f"{spherical.rank() = }")
```

```
cartesian.rank() = 1
spherical.rank() = 1
```

4.4 Defining Coordinate Transformations

The next logical step after defining our Coordinate systems is adding the rules for transforming between them. We can let PyOGRe know how to transform coordinates by adding the transformation rules using the `add_coord_transformation()` method on any Coordinates object. The first argument **coords** is the target coordinate system, and the second argument **rules** is a list of the transformation rules. The rules must be a list of SymPy equations of the form $x_i = f(\vec{x})$, where x_i is one of the source coordinates, and $f(\vec{x})$ is a function of the target coordinates. If we leave the argument rules blank, PyOGRe will remove any existing transformation rules to the target coordinate system.

[ ]:
```
print(cartesian.add_coord_transformation.__doc__)
```

```
        (method) add_coord_transformation(coords, rules)

        Define a Coordinate transformation.

        `coords`: The target coordinate system.
        `rules`: The transformation rules to convert the current coordinates to
the target coordinates.
        If left as None, and coords is supplied, any existing transformation
will be removed.

        Example:

        >>> import sympy as sym
        >>> from PyOGRe.Coordinates import new_coordinates

        >>> t, x, y, z, r, theta, phi = sym.symbols("t x y z r theta phi")

        >>> cartesian = new_coordinates(
            name="4D Cartesian",
            components=sym.Array([t, x, y, z])
        )

        >>> spherical = new_coordinates(
            name="4D Spherical",
            components=sym.Array([t, r, theta, phi])
        )

        >>> cartesian.add_coord_transformation(
            coords=spherical,
```

```
            rules=[
                None,
                sym.Eq(x, r*sym.sin(theta)*sym.cos(phi)),
                sym.Eq(y, r*sym.sin(theta)*sym.sin(phi)),
                sym.Eq(z, r*sym.cos(theta))
            ]
        )

        >>> spherical.add_coord_transformation(
            coords=cartesian,
            rules=[
                None,
                sym.Eq(r, sym.sqrt(x**2+y**2+z**2)),
                sym.Eq(theta, sym.acos(z/(sym.sqrt(x**2+y**2+z**2)))),
                sym.Eq(phi, sym.atan(y/x)),
            ]
        )
```

Above, we defined both the Cartesian and Spherical coordinate systems in 4D. We then
added the coordinate transformations between the two systems.

[ ]:
```
# t -> t
# x -> r*sin(theta)*cos(phi)
# y -> r*sin(theta)*sin(phi)
# z -> r*cos(theta)
cartesian_to_spherical = [
    sym.Eq(x, r*sym.sin(theta)*sym.cos(phi)),
    sym.Eq(y, r*sym.sin(theta)*sym.sin(phi)),
    sym.Eq(z, r*sym.cos(theta))
]

cartesian.add_coord_transformation(
    coords=spherical,
    rules=cartesian_to_spherical
)
```

[ ]: Cartesian

[ ]:
```
# t -> t
# r -> sqrt(x^2+y^2+z^2)
# theta -> acos(z/(sqrt(x^2+y^2+z^2)))
# phi -> atan(y/x)
spherical_to_cartesian = [
    sym.Eq(r, sym.sqrt(x**2+y**2+z**2)),
    sym.Eq(theta, sym.acos(z/(sym.sqrt(x**2+y**2+z**2)))),
    sym.Eq(phi, sym.atan(y/x)),
```

```
]

spherical.add_coord_transformation(
    coords=cartesian,
    rules=spherical_to_cartesian
)
```

[ ]:
```
Spherical
```

Above, we have defined the transformation rules to go from Cartesian to Spherical coordinates, as well as the inverse transformation. As seen in the examples, if one coordinate does not change during the transformation, we can simply omit the rule (or you can supply `None` as the rule).

4.5 Defining Metrics

With our coordinate systems defined, we can now create metric tensors in order to define our manifolds. In PyOGRe, we can use the `new_metric()` method on a Coordinates object (or the equivalent standalone function `new_metric()`) to create a new metric tensor.

[ ]:
```
print(og.new_metric.__doc__)
```

```
    (function) new_metric(name, components, coords, indices, symbol)

    Creates a new tensor object representing a metric tensor.

    `name`: Defines name of the tensor is used when displaying the metric.
    `components`: An array / matrix of the metric's components.
    `coords`: Defines the coordinates the metric is using.
    `indices`: Defines the default indices of the metric, must be (1, 1) or (-1,
-1).
    `symbol`: Defines the symbol used to represent the metric object.

    Example:

    >>> import sympy as sym
    >>> from PyOGRe import new_coordinates, new_metric
    >>> cartesian = new_coordinates("Cartesian", sym.Array([sym.Symbol("t"),
sym.Symbol("x"), sym.Symbol("y"), sym.Symbol("z")]))
    >>> new_metric("Minkowski", sym.diag(-1, 1, 1, 1), cartesian)

    This will create the standard 3+1 dimensional Minkowski metric defined in
standard Cartesian coordinates.
```

Just like when we defined the Coordinates objects, when defining a metric we must supply the **name** of the metric tensor, and the **components**. If we use the standalone function as we do below, we must also supply the argument **coords** to let PyOGRe know which coordinate system the metric is using. It should be noted that by default, PyOGRe assumes the components of the

metric are being supplied with two lower indices, but this can be overwritten by overwriting the argument **indices**. Again like Coordinates objects, a **symbol** may be supplied which will be used when displaying the metric, but by default PyOGRe will use g.

[ ]:
```
minkowski = og.new_metric(
    name="Minkowski",
    coords=cartesian,
    components=sym.diag(-1, 1, 1, 1),
    symbol="eta"
)
minkowski.show()
```

Minkowski:

$$\eta_{\mu\nu}\left(t,x,y,z\right) = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Above we have defined the standard 3+1 dimenstional Minkwoski metric using the Cartesian coordinates we defined earlier. As can be seen, we manually had to supply the coordinates when using the standalone function `new_metric()`. We can just as easily define the Schwarzschild metric in Spherical coordinates by using the `new_metric()` method on the `spherical` Coordinates object, as can be seen in the following example.

[ ]:
```
M = sym.symbols("M")

schwarzschild = spherical.new_metric(
    name="Schwarzschild",
    components=sym.Array(
        [
            [-(1-2*M/r), 0, 0, 0],
            [0, 1/(1-2*M/r), 0, 0],
            [0, 0, r**2, 0],
            [0, 0, 0, r**2 * sym.sin(theta)**2]
        ]
    )
)
schwarzschild.show()
```

Schwarzschild:

$$g_{\mu\nu}\left(t,r,\theta,\phi\right) = \begin{pmatrix} \frac{2M}{r} - 1 & 0 & 0 & 0 \\ 0 & \frac{1}{-\frac{2M}{r}+1} & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2 \sin^2\left(\theta\right) \end{pmatrix}$$

4.5.1 Displaying Line Elements

With our metrics created, we can now ask PyOGRe to display them as a line element using the `line_element()` method.

[ ]:
```
minkowski.line_element()
minkowski.line_element(coords=spherical)
```

Minkowski:

$$\mathrm{d}s^2 = -\mathrm{d}t^2 + \mathrm{d}x^2 + \mathrm{d}y^2 + \mathrm{d}z^2$$

Minkowski:

$$\mathrm{d}s^2 = -\mathrm{d}t^2 + \mathrm{d}r^2 + r^2\mathrm{d}\theta^2 + r^2\sin^2(\theta)\mathrm{d}\phi^2$$

4.5.2 Displaying Volume Elements

Similarly to the line elements, we can display the volume element squared of a metric using the `volume_element()` method, which is just the determinant of the metric.

[ ]:
```
minkowski.volume_element()
minkowski.volume_element(coords=spherical)
```

Minkowski:

$$\mathrm{d}V^2 = -1$$

Minkowski:

$$\mathrm{d}V^2 = -r^4\sin^2(\theta)$$

4.6 Defining Tensors

Any tensors that are not coordinates or a metric can be defined using the **`new_tensor()`** method on any metric, or using the standalone function **`new_tensor()`**. If we use the method, PyOGRe will automatically associate the new tensor with the metric the method was called on, and assume that the tensor is defined using the same coordinates the metric uses by default (although this can be overwritten by specifying the argument **coords**).

[ ]:
```
print(og.new_tensor.__doc__)
```

```
(function) new_tensor(name, components, metric, coords, indices, symbol)

Creates a new tensor object representing a tensor.

`name`: Defines name of the tensor is used when displaying the tensor.
`components`: An array / matrix of the tensor's components.
```

```
    `metric`: Defines the metric tensor associated with the tensor.
    `coords`: Defines the coordinates the tensor is using.
    `indices`: Defines the default indices of the tensor, each index must be 1
(contravariant) or -1 (covariant).
    `symbol`: Defines the symbol used to represent the metric object.

    Example:

    >>> import sympy as sym
    >>> from PyOGRe import new_coordinates, new_metric, new_tensor
    >>> cartesian = new_coordinates("Cartesian", sym.Array([sym.Symbol("t"),
sym.Symbol("x"), sym.Symbol("y"), sym.Symbol("z")]))
    >>> minkowski = new_metric("Minkowski", sym.diag(-1, 1, 1, 1), cartesian)
    >>> og.new_tensor("Scalar", sym.Array(42), minkowski, cartesian, (), "S")

    This will create the scalar 42 in the Minkowski metric and Cartesian
coordinates.
```

4.6.1 Scalars

To start, let us first create the Kretschmann scalar in the Schwarzschild spacetime (we will later show how to calculate it from the metric using PyOGRe).

[ ]:
```
kretschmann = schwarzschild.new_tensor(
    name="Kretschmann",
    components=sym.Array(48*M**2/r**6),
    indices=(),
    symbol="S"
)
kretschmann.show()
```

Kretschmann:

$$S\left(t,r,\theta,\phi\right)=\frac{48M^{2}}{r^{6}}$$

4.6.2 Generic Tensors

Just as for scalar quantities, we can create arbitrary tensors of any rank using the same method or function `new_tensor()`. For example, let us create a vector (a rank 1 tensor) in the Minkowski spacetime representing the velocity of a particle travelling in the x direction with velocity v.

[ ]:
```
v = sym.symbols("v")

four_velocity = og.new_tensor(
    name="4-Velocity",
    components=1/sym.sqrt(1-v**2) * sym.Array([1, v, 0, 0]),
    indices=(1,),
```

```
    coords=cartesian,
    metric=minkowski,
    symbol="u"
)
four_velocity.show()
```

4-Velocity:

$$u^{\mu}\left(t,x,y,z\right)=\begin{pmatrix}\frac{1}{\sqrt{1-v^2}}\\ \frac{v}{\sqrt{1-v^2}}\\ 0\\ 0\end{pmatrix}$$

As another example, let us create a rank 2 tensor in the Minkowski spacetime. We will define the stress-energy tensor for a perfect fluid by using the matrix representation with two upper indices.

[ ]:
```
rho, p = sym.symbols("rho p")

perfect_fluid = minkowski.new_tensor(
    name="Perfect Fluid",
    components=sym.diag(rho, p, p, p),
    indices=(1, 1),
    symbol="T"
)
perfect_fluid.show()
```

Perfect Fluid:

$$T^{\mu\nu}\left(t,x,y,z\right)=\begin{pmatrix}\rho & 0 & 0 & 0\\ 0 & p & 0 & 0\\ 0 & 0 & p & 0\\ 0 & 0 & 0 & p\end{pmatrix}$$

We could just as easily continue with higher rank tensors, but we will not go into that here as higher rank tensors are often derived in calculations. Later, we will see how to perform such calculations using PyOGRe to derive quantities such as the Christoffel symbols or the Riemann tensor from the metric (which are rank 3 and rank 4 tensors respectively).

4.7 Transforming Tensors

With our tensors defined, we may want to start requesting the tensors with a different index configuration. In order to change the index configuration, we must raise and lower the indices by **contracting** the tensor with the metric. As an example, suppose we have a vector v^{ν} represented with one upper index and a metric g. We can lower the index, turning the vector into a covector as follows:

$$v_\mu = g_{\mu\nu} v^\nu$$

Here, we have lowered one index. In the example above, as well as the rest of the documentation, we will be using the **Einstein summation convention**, which states that whenever the same index is repeated exactly twice, once as an upper index and once as a lower index, it is to be summed over. In the example above, the summation is over $\nu \in \{0, 1, 2, 3\}$, and writing the summation out yields:

$$v_\mu = \sum_{\nu=0}^{3} g_{\mu\nu} v^\nu = g_{\mu 0} v^0 + g_{\mu 1} v^1 + g_{\mu 2} v^2 + g_{\mu 3} v^3$$

Sums such as the one above are what is known as a **contraction**, which is simply a generalization of the familiar inner product. If we instead had a covector w_μ, we could raise the index turning it into a vector by contracting it with the inverse metric:

$$w^\mu = g^{\mu\nu} w_\nu.$$

Of course, this can be extended to work for tensors of any rank. For example, suppose we have a tensor $T_a{}^b$ with one upper index and one lower index. We can raise or lower either index, or invert the index configuration:

$$T^{ab} = g^{a\lambda} T_\lambda{}^b, \quad T_{ab} = g_{b\lambda} T_a{}^\lambda, \quad T^a{}_b = g_{b\rho} g^{a\lambda} T_\lambda{}^\rho.$$

However, when using PyOGRe one doesn't need to worry about ever raising or lowering an index. The tensor objects are **abstract tensors** which are completely free of any single index configuration. All one must do instead is request PyOGRe to display the components of the tensor with the desired index configuration and the desired coordinate system. Both the `show()` and `list()` methods in PyOGRe will accept and argument **coords** to specify the coordinate system to use when displaying the tensor, along with the argument **indices** which will be used to request specific index configurations.

Additionally, it may become useful to start displaying them in different coordinate systems than the one they were defined in. To understand how we transform coordinates, let us assume we have defined a tensor in a coordinate system x^μ. Then we can transform it to a different coordinate system $x^{\mu'}$ in the following way:

- For each lower index, add a factor of $\dfrac{\partial x^\mu}{\partial x^{\mu'}}$. This is the derivative of the source coordinates with respect to the target coordinates, which is also known as the **Jacobian**.
- For each upper index, add a factor of $\dfrac{\partial x^{\mu'}}{\partial x^\mu}$. This is the derivative of the target coordinates with respect to the source coordinates, which is also known as the **Inverse Jacobian**.

As an example, consider the tensor T_{ab} defined in coordinates x^μ. Transforming coordinates to a new coordinate system $x^{\mu'}$ will result in $T_{a'b'}$ given by:

$$T_{a'b'}(x^{\mu'}) = \frac{\partial x^a}{\partial x^{a'}} \frac{\partial x^b}{\partial x^{b'}} T_{ab}(x^{\mu})$$

In general, the above rules can be summarized for a tensor of type (p, q) that has p upper indices $a_1, \ldots, a_p$ and q lower indices $b_1, \ldots, b_q$, such that the transformation from x^{μ} to $x^{\mu'}$ is given by:

$$T^{a'_1 \ldots a'_p}_{b'_1 \ldots b'_q}(x^{\mu'}) = \left(\frac{\partial x^{a'_1}}{\partial x^{a_1}} \cdots \frac{\partial x^{a'_p}}{\partial x^{a_p}} \right) \left(\frac{\partial x^{b_1}}{\partial x^{b'_1}} \cdots \frac{\partial x^{b_q}}{\partial x^{b'_q}} \right) T^{a_1 \ldots a_p}_{b_1 \ldots b_q}(x^{\mu})$$

Any easy way to remember the above formula is to recall that two indices may only appear twice in any formula, once as an upper index and once as a lower index so that it may be contracted properly. Thus, the indices can only be placed in one logical position; that is, if a_1 is an upper index in T, then it must be contracted with a lower index, so ∂x^{a_1} must appear in the denominator of the transformation.

For our first example, let us display the inverse of the Minkowski metric, by raising both of the indices:

[ ]:
```
minkowski.list(indices=(1, 1))
```

Minkowski:

$$\eta^{tt} = -\eta^{xx} = -\eta^{yy} = -\eta^{zz} = -1$$

Or instead, we could ask to see the components of the Minkowski metric in spherical coordinates:

[ ]:
```
minkowski.list(coords=spherical)
```

Minkowski:

$$\eta_{tt} = -\eta_{rr} = -1$$

$$\eta_{\theta\theta} = r^2$$

$$\eta_{\phi\phi} = r^2 \sin^2(\theta)$$

And why restrict ourselves to only one option? We can request both a different coordinate system and index configuration in the same method call:

[ ]:
```
minkowski.list(coords=spherical, indices=(1, 1))
```

Minkowski:

$$\eta^{tt} = -\eta^{rr} = -1$$

$$\eta^{\theta\theta} = \frac{1}{r^2}$$

$$\eta^{\phi\phi} = \frac{1}{r^2 \sin^2(\theta)}$$

4.8 Performing Substitutions and Mapping Functions

Along with requesting the representation of a tensor in a different coordinate system or index configuration, PyOGRe also allows one to perform substitutions and map a function to each component of the tensor. Each of these are done by passing the keyword argument **replace** and the keyword argument **function** respectively. As an example, suppose we wanted to list the components of the Minkowski metric in spherical coordinates, but with the radial coordinate evaluated at $r = 2$ and the polar angle evaluated at $\theta = \pi/2$:

[ ]:
```
minkowski.list(coords=spherical, indices=(1, 1), replace={r: 2, theta: sym.pi/
↪4})
```

Minkowski:

$$\eta^{tt} = -\eta^{rr} = -1$$

$$\eta^{\theta\theta} = \frac{1}{4}$$

$$\eta^{\phi\phi} = \frac{1}{2}$$

As we can see above, replace accepts a dictionary of substitutions where the keys are SymPy expressions, and the values are the values we would like to substitute in for the keys. If we then wanted to take the absolute value of the components above, we could pass the `abs()` function as the **function** argument:

[ ]:
```
minkowski.list(coords=spherical, indices=(1, 1), replace={r: 2, theta: sym.pi/
↪4}, function=abs)
```

Minkowski:

$$\eta^{tt} = \eta^{rr} = 1$$

$$\eta^{\theta\theta} = \frac{1}{4}$$

$$\eta^{\phi\phi} = \frac{1}{2}$$

It should be noted that there is also a keyword argument called **args** which can be used to pass additional arguments to the **function**.

4.9 Retrieving Tensor Components

We now know how to display tensor components, but what about retrieving them so that we could perform our own calculations with them? This is handled by the `get_components()` method. This method supports a keyword argument **mode** which allows the user to specify whether the components should be returned as a SymPy array (the default), as a Mathematica array, or as a LaTeX expression.

4.9.1 As a SymPy Array

To start, we can get the components of any tensor object in PyOGRe as a SymPy array simply by calling the `get_components()` method. The method accepts the same arguments as both `list()` and `show()`, so we can easily get the components in any coordinate system, index configuration, or with substitutions made.

[ ]:
```
sympy_coordinates = cartesian.get_components()
print(sympy_coordinates)

sympy_minkowski = minkowski.get_components(coords=spherical, indices=(-1, -1))
print(sympy_minkowski)

sympy_minkowski = minkowski.get_components(coords=spherical, indices=(-1, -1),␣
↪mode="sympy")
print(sympy_minkowski)
```

```
[t, x, y, z]
[[-1, 0, 0, 0], [0, 1, 0, 0], [0, 0, r**2, 0], [0, 0, 0, r**2*sin(theta)**2]]
[[-1, 0, 0, 0], [0, 1, 0, 0], [0, 0, r**2, 0], [0, 0, 0, r**2*sin(theta)**2]]
```

4.9.2 As a Mathematica List

To request the components of a tensor as a Mathematica list, we can pass **mode="mathematica"** as an argument to the `get_components()` method.

[ ]:
```
mathematica_coordinates = cartesian.get_components(mode="mathematica")
print(mathematica_coordinates)

mathematica_minkowski = minkowski.get_components(coords=spherical, indices=(-1,␣
↪-1), mode="mathematica")
print(mathematica_minkowski)
```

```
{t, x, y, z}
{{-1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, r^2, 0}, {0, 0, 0, r^2*Sin[theta]^2}}
```

4.9.3 As a LaTex Expression

Just as we did above, we can also pass **mode="latex"** to get the components of a tensor as a LaTeX expression.

[ ]:
```
latex_coordinates = cartesian.get_components(mode="latex")
print(latex_coordinates)

latex_minkowski = minkowski.get_components(coords=spherical, indices=(-1, -1),␣
↪mode="latex")
print(latex_minkowski)
```

```
\left(\begin{matrix}t\\x\\y\\z\end{matrix}\right)
\left(\begin{matrix}-1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & r^{2} & 0\\0 & 0 & 0
& r^{2} \sin^{2}{\left(\theta \right)}\end{matrix}\right)
```

4.10 Changing Defaults

4.10.1 Setting the Index Letters

By default, PyOGRe uses the Greek alphabet to label the indices of a tensor when displaying the components. In order to display the current letters used for the indices, or to change them, we can use the `set_index_letters()` function.

[ ]:
```
print(og.set_index_letters.__doc__)
```

```
    This function will either print the existing index letters, or will
overwrite them if an argument is given.

    Supplying an argument of 'automatic' will set the index letters to the
default.

    Expects the argument 'letters' to be of the form 'a b c d…'.

    >>> og.set_index_letters("a b c d e f g h i j k l m n o p")
    (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p)

    >>> og.set_index_letters("automatic")
    (mu, nu, rho, sigma, kappa, lambda, alpha, beta, gamma, delta, epsilon,
zeta, theta, iota, xi, pi, tau, phi, chi, psi, omega)
```

[ ]:
```
og.set_index_letters()
og.set_index_letters("a b c d e f g h i j k l m n o p")
og.set_index_letters("automatic")
```

$(\mu,\ \nu,\ \rho,\ \sigma,\ \kappa,\ \lambda,\ \alpha,\ \beta,\ \gamma,\ \delta,\ \epsilon,\ \zeta,\ \theta,\ \iota,\ \xi,\ \pi,\ \tau,\ \phi,\ \chi,\ \psi,\ \omega)$

$(a,\ b,\ c,\ d,\ e,\ f,\ g,\ h,\ i,\ j,\ k,\ l,\ m,\ n,\ o,\ p)$

$(\mu,\ \nu,\ \rho,\ \sigma,\ \kappa,\ \lambda,\ \alpha,\ \beta,\ \gamma,\ \delta,\ \epsilon,\ \zeta,\ \theta,\ \iota,\ \xi,\ \pi,\ \tau,\ \phi,\ \chi,\ \psi,\ \omega)$

4.10.2 Changing the Name of a Tensor

If we would like to change the name of a tensor, we can call the `change_name()` method.

[ ]:
```
four_velocity.change_name("Four Velocity")
four_velocity.info()
```

Four Velocity:

Name: Four Velocity

Symbol: u

Type: Tensor

Rank: 1

Metric: Minkowski

Default Coordinates: Cartesian

Default Indices: (1,)

4.10.3 Changing the Tensor Symbol

Similarly, we can change the symbol used to represent the tensor by calling the `change_symbol()` method.

[ ]:
```
kretschmann.change_symbol("K")
kretschmann.show()
```

Kretschmann:

$$K\left(t, r, \theta, \phi\right) = \frac{48M^2}{r^6}$$

4.10.4 Changing the Default Indices

When we define a tensor in PyOGRe, PyOGRe automatically takes the supplied index configuration as the **default index configuration**. This will be the index confuration used when `list()` or `show()` are called without specifying the **indices** argument. If we would like to change the default index configuration, we can call the `change_default_indices()` method.

[ ]:
```
four_velocity.change_default_indices(indices=(-1,))
four_velocity.list()
```

Four Velocity:

$$u_t = -\frac{1}{\sqrt{1-v^2}}$$

$$u_x = \frac{v}{\sqrt{1-v^2}}$$

4.10.5 Changing the Default Coordinates

Similarly to the default index configuration, the coordinates used to define a tensor become the tensor's **default coordinates**, and just like before we can change the default coordinates by calling

the `change_default_coords()` method. If the tensor cannot be transformed to the new coordinates though, this method will fail.

[ ]:
```
four_velocity.change_default_coords(coords=spherical)
four_velocity.list()
```

Four Velocity:

$$u_t = -\frac{1}{\sqrt{1-v^2}}$$

$$u_r = \frac{v\sin(\theta)\cos(\phi)}{\sqrt{1-v^2}}$$

$$u_\theta = \frac{rv\cos(\phi)\cos(\theta)}{\sqrt{1-v^2}}$$

$$u_\phi = -\frac{rv\sin(\phi)\sin(\theta)}{\sqrt{1-v^2}}$$

[ ]:
```
four_velocity.change_default_coords(coords=cartesian).show()
```

Four Velocity:

$$u_\mu(t,x,y,z) = \begin{pmatrix} -\frac{1}{\sqrt{1-v^2}} \\ \frac{v}{\sqrt{1-v^2}} \\ 0 \\ 0 \end{pmatrix}$$

4.11 Deleting PyOGRe Objects

We can delete PyOGRe objects by calling the `delete()` method. This method will remove the object from the PyOGRe workspace only if the tensor being deleted is not used in the definition of any other tensor, for example as a coordinate system. As an example, we can create a temporary coordinate system. We can then delete it without error because we have not created any metrics or tensors that depend on this new coordinate system.

[ ]:
```
bad_coordinates = og.new_coordinates(
    name="I was a mistake",
    components=cartesian_symbols
)

bad_coordinates.delete()
```

I was a mistake:

Successfully Deleted

4.12 Retrieving All Tensor Objects

To get a list of all currently defined PyOGRe objects, we can use the `get_instances()` function. Calling this function will return a list of all tensors.

[ ]:
```
og.get_instances()
```

[ ]:
```
[Cartesian,
 Spherical,
 Minkowski,
 Schwarzschild,
 Kretschmann,
 Four Velocity,
 Perfect Fluid]
```

4.13 Cleaning Up Result Tensors

As we will see later, any tensors created from calculations in PyOGRe will by default get named "Result". Should we like to delete any such tensors, we can use the `delete_results()` function.

[ ]:
```
og.delete_results()
display(og.get_instances())
```

Deleted 0 tensor(s) named 'Result'.

```
[Cartesian,
 Spherical,
 Minkowski,
 Schwarzschild,
 Kretschmann,
 Four Velocity,
 Perfect Fluid]
```

5 Importing and Exporting Tensors

5.1 Exporting

A key feature of PyOGRe is the ability to import and export tensors, and specifically import and export tensors between the Mathematica and Python versions of OGRe. To start, any tensor in PyOGRe can be exported by calling the `export()` method. This method will return a string which can later be used to import the tensor in either version of OGRe, or should you wish to export it to a file you can supply a filepath as an argument.

[ ]:
```
# print(minkowski.export())
```

[ ]:
```
cartesian.export("cartesian.txt")
```

Exported Cartesian to cartesian.txt

If we instead wanted to export **all** tensors currently available, we can use the `export_all()` function. Just like the `export()` method, this function will return a string which can later be used

to import the tensors in either version of OGRe, or should you wish to export it to a file you can supply a filepath as an argument.

[ ]:
```
# print(og.export_all())
```

[ ]:
```
og.export_all("tensors.txt")
```

Exported all tensors to tensors.txt

5.2 Importing

Given a string containing a tensor obtained either from exporting a tensor in PyOGRe or in OGRe, we can import it by calling the `import_from_string()` function. The function will simply return the new PyOGRe object.

[ ]:
```
og.import_from_string(
'<|"Imported Minkowski"-><|"Components"-><|{{-1,-1},"Cartesian"}->{{-1, 0, 0,␣
 ↪0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}},{{1,-1},"Cartesian"}->{{1, 0,␣
 ↪0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}},{{-1,1},"Cartesian"}->{{1,␣
 ↪0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0,␣
 ↪1}},{{1,1},"Cartesian"}->{{-1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0,␣
 ↪0, 1}},{{-1,-1},"Spherical"}->{{-1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, r^2, 0},␣
 ↪{0, 0, 0, r^2*Sin[theta]^2}},{{1,1},"Spherical"}->{{-1, 0, 0, 0}, {0, 1, 0,␣
 ↪0}, {0, 0, r^(-2), 0}, {0, 0, 0, 1/
 ↪(r^2*Sin[theta]^2)}},{{-1,1},"Spherical"}->{{1, 0, 0, 0}, {0, 1, 0, 0}, {0,␣
 ↪0, 1, 0}, {0, 0, 0, 1}},{{1,-1},"Spherical"}->{{1, 0, 0, 0}, {0, 1, 0, 0},␣
 ↪{0, 0, 1, 0}, {0, 0, 0,␣
 ↪1}}|>,"DefaultCoords"->"Cartesian","DefaultIndices"->{-1,␣
 ↪-1},"Role"->"Metric","Symbol"->g,"Metric"->"Minkowski","OGReVersion"->"PyOGRe␣
 ↪v0.0.1"|>|>'
).info()
```

Imported Minkowski:

Name: Imported Minkowski

Symbol: g

Type: Metric

Rank: 2

Default Coordinates: Cartesian

Default Indices: (-1, -1)

Tensors Using This Metric: Imported Minkowski

To import a tensor from a file containing a tensor, we can call the `import_from_file()` function. Just as before, the function will simply return the newly imported tensor object.

[ ]:
```
og.import_from_file("cartesian.txt").change_name("Imported Cartesian").show()
```

Imported Cartesian:

$$x^\mu = \begin{pmatrix} t \\ x \\ y \\ z \end{pmatrix}$$

If we would instead like to import all tensors from a file or string, we can use either the `import_all_from_file()` or `import_all_from_string` functions respectively. These functions will return a list of all tensors imported from the file or string, but be warned, they will also **delete all current tensors** that are currently available.

[ ]:
```
# tensors_list = og.import_all_from_file("tensors.txt")
```

6 Built-in Tensors

PyOGRe by default includes some common coordinates and metrics which can simply be imported for use. Currently, these are:

- $3+1$ Cartesian coordinates (t, x, y, z)
- $3+1$ Spherical coordinates (t, r, θ, ϕ)
- The Minkowski metric with signature $(-, +, +, +)$
- The Schwarzschild metric
- The Friedmann–Lemaître–Robertson–Walker (FLRW) metric

Note: The Cartesian and Spherical coordinates systems already have the coordinate transformations defined in both directions.

[ ]:
```
from PyOGRe.Defaults import cartesian as default_cartesian
from PyOGRe.Defaults import spherical as default_spherical
from PyOGRe.Defaults import minkowski as default_minkowski
from PyOGRe.Defaults import schwarzschild as default_schwarzschild
from PyOGRe.Defaults import flrw as default_flrw
```

[ ]:
```
default_cartesian.show()
default_spherical.show()
default_minkowski.show()
default_schwarzschild.show()
default_flrw.show()
```

4D Cartesian:

$$x^\mu = \begin{pmatrix} t \\ x \\ y \\ z \end{pmatrix}$$

4D Spherical:

$$x^{\mu} = \begin{pmatrix} t \\ r \\ \theta \\ \phi \end{pmatrix}$$

4D Minkowski:

$$g_{\mu\nu}\left(t, x, y, z\right) = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Schwarzschild:

$$g_{\mu\nu}\left(t, r, \theta, \phi\right) = \begin{pmatrix} \frac{2M}{r} - 1 & 0 & 0 & 0 \\ 0 & \frac{1}{-\frac{2M}{r}+1} & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2 \sin^2\left(\theta\right) \end{pmatrix}$$

FLRW:

$$g_{\mu\nu}\left(t, r, \theta, \phi\right) = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & \frac{a^2(t)}{-kr^2+1} & 0 & 0 \\ 0 & 0 & r^2 a^2(t) & 0 \\ 0 & 0 & 0 & r^2 a^2(t) \sin^2\left(\theta\right) \end{pmatrix}$$

7 Performing Tensor Calculations

Now that we know how to create and manage tensor objects using PyOGRe, we can introduce the ability to perform calculations with tensors. Performing any calculation in PyOGRe is done by calling the `Calc()` function.

[ ]:
```
print(og.Calc.__doc__)
```

```
(function) Calc(calc_object, name, symbol, indices)

Calculates a tensor formula.

`calc_object`: The tensor formula.
`name`: Defines the name of the resulting tensor. Defaults to "Result".
`symbol`: Defines the symbol used to represent the resulting tensor. A
```

```
placeholder symbol will be used by default.
    `indices`: A string representing the order of indices of the resulting
tensor. Defaults to the order that appears in the tensor formula.
```

The result of calling `Calc()` will be another tensor object. The formula used to calculate the resulting tensor may include any combination of addition, scalar multiplication, trace, contraction, partial derivatives, and covariant derivatives. We will cover each of these operations in more detail below, along with some examples.

7.1 Simplifying Tensors

To start, we can note that any tensor can be simplified by calling the `simplify()` method. This will not return a new object, but will simplify all known representations of the tensor, and return itself.

[ ]:
```
minkowski.simplify().show()
```

```
Simplifying:   0%|          | 0/128 [00:00<?, ?it/s]
```

Minkowski:

$$\eta_{\mu\nu}\left(t,x,y,z\right)=\begin{pmatrix}-1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}$$

7.2 Addition and Subtraction of Tensors

Addition of tensors in PyOGRe is done by calling each tensor with a string of indices, and adding the resulting objects. When performing tensor addition we also have a handful of constraints to keep in mind:

- Coordinates cannot be added to any other tensor, as they do not transform like tensors.
- Two tensors that use different metrics may not be added together, as their sum would not have well-defined transformation rules.
- Adding two tensors of different rank is not possible.
- The indices of both tensors must be the same up to permutation:
  - $M^{ab}+N^{ab}$ or $M^{ab}+N^{ba}$ are both allowed,
  - $M^{ab}+N^{ac}$ or $M^{ab}+N^{cb}$ are not.

This all sounds more complicated than it actually is, so let us start with an example:

[ ]:
```
og.Calc(
    minkowski("mu nu") + perfect_fluid("mu nu"),
).show(replace={v: 0})
```

Result:

$$\square_{\mu\nu}(t,x,y,z) = \begin{pmatrix} \rho - 1 & 0 & 0 & 0 \\ 0 & p+1 & 0 & 0 \\ 0 & 0 & p+1 & 0 \\ 0 & 0 & 0 & p+1 \end{pmatrix}$$

Here, we have added the Minkowski metric with the perfect fluid tensor we defined earlier (and requested that we show the result with $v = 0$). The tensor formula is written as `tensor1("indices1") + tensor2("indices2")`. In this expression, `tensor1` and `tensor2` are the two PyOGRe objects we wish to add, and `"indices1"` and `"indices2"` are the index specifications for each tensor as a string. We should note that we don't need to specify whether each index is an upper or lower index, as PyOGRe deduces this automatically. Of course, the index letters have no real meaning and are simply placeholders, however we must remain cautious and ensure the indices are consistent.

Returning to our example, we can pass the argument **symbol** to replace the default symbol for the resulting tensor, as well as the argument **name** to give our result a name.

[ ]:
```
og.Calc(
    minkowski("mu nu") + perfect_fluid("mu nu"),
    symbol="S",
    name="Sum Result"
).show(replace={v: 0})
```

Sum Result:

$$S_{\mu\nu}(t,x,y,z) = \begin{pmatrix} \rho - 1 & 0 & 0 & 0 \\ 0 & p+1 & 0 & 0 \\ 0 & 0 & p+1 & 0 \\ 0 & 0 & 0 & p+1 \end{pmatrix}$$

It can sometimes be useful to specify the resulting index order, which we can demonstrate in the following example. First, we will define a simple non-symmetric tensor:

[ ]:
```
non_symmetric = minkowski.new_tensor(
    name="Non-Symmetric",
    components=sym.Array([[0, 0, 0, 1], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0,␣
 ↪0]]),
    symbol="N",
    indices=(-1, -1)
)
non_symmetric.show()
```

Non-Symmetric:

$$N_{\mu\nu}(t,x,y,z) = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Now, let us compare the result of adding our non-symmetric tensor and the Minkowski metric in the following ways:

1) $\eta_{\mu\nu} + N_{\mu\nu}$
2) $\eta_{\mu\nu} + N_{\nu\mu}$

First, with the same index order, we get:

[ ]:
```
og.Calc(
    minkowski("mu nu") + non_symmetric("mu nu")
).show()
```

Result:

$$\square_{\mu\nu}\,(t,x,y,z) = \begin{pmatrix} -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

And with the index order for the non-symmetric tensor reversed:

[ ]:
```
og.Calc(
    minkowski("mu nu") + non_symmetric("nu mu")
).show()
```

Result:

$$\square_{\mu\nu}\,(t,x,y,z) = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

Since the order of the indices matters, we may want to explicitly specify the index order for the resulting tensor. We can pass the argument **indices** to indicate the order we want:

[ ]:
```
og.Calc(
    minkowski("mu nu") + non_symmetric("nu mu"),
    indices="nu mu"
).show()
```

Result:

$$\square_{\mu\nu}\,(t,x,y,z) = \begin{pmatrix} -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

As you can see, we have recovered the original result by specifying the order of the resulting tensor's indices. Finally, we can also choose to add more than just two tensors together, so long as we keep the indices consistent. For example, we can calculate the following sum:

[ ]:
```
og.Calc(
    minkowski("mu nu") + perfect_fluid("mu nu") + non_symmetric("mu nu") +␣
 ↪non_symmetric("nu mu")
).show(replace={v: 0})
```

Result:

$$\square_{\mu\nu}\left(t,x,y,z\right) = \begin{pmatrix} \rho - 1 & 0 & 0 & 1 \\ 0 & p + 1 & 0 & 0 \\ 0 & 0 & p + 1 & 0 \\ 1 & 0 & 0 & p + 1 \end{pmatrix}$$

7.3 Multiplication of Tensor by a Scalar

The next operation one can perform with tensors is naturally the multiplication of a tensor by a scalar. The form of a tensor formula involving multiplication by a scalar is `scalar * tensor("indices")`, where the `scalar` can be any SymPy expression, `tensor` is the tensor object, and `"indices"` is an index specification as it was for tensor addition. Additionally, `scalar` may also be a tensor object of rank 0.

As an example, let us simply multiply the Minkowski metric by 10:

[ ]:
```
og.Calc(
    10 * minkowski("a b")
).show()
```

Result:

$$\square_{\mu\nu}\left(t,x,y,z\right) = \begin{pmatrix} -10 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 10 \end{pmatrix}$$

For an example with a tensor of rank 0, let us multiply the Schwarzschild metric by the Kretschmann:

[ ]:
```
og.Calc(
    kretschmann() * schwarzschild("mu nu")
).show()
```

Result:

$$\square_{\mu\nu}\left(t,r,\theta,\phi\right) = \begin{pmatrix} \frac{48M^2 \cdot (2M - r)}{r^7} & 0 & 0 & 0 \\ 0 & \frac{48M^2}{r^5(-2M + r)} & 0 & 0 \\ 0 & 0 & \frac{48M^2}{r^4} & 0 \\ 0 & 0 & 0 & \frac{48M^2 \sin^2\left(\theta\right)}{r^4} \end{pmatrix}$$

At first glance, specifying the indices for the tensor may seem unnecessary, but we must remember that we would ultimately like to combine multiplication by a scalar with other operations, in which case the indices become necessary, as is demonstrated in the following example.

[ ]:
```
og.Calc(
    2 * t * minkowski("mu nu") - 3 * x * perfect_fluid("mu nu") + 4 * y *␣
 ↪non_symmetric("mu nu") - 5 * z * non_symmetric("nu mu")
).show(replace={v: 0})
```

Result:

$$\Box_{\mu\nu}\left(t,x,y,z\right) = \begin{pmatrix} -3\rho x - 2t & 0 & 0 & 4y \\ 0 & -3px + 2t & 0 & 0 \\ 0 & 0 & -3px + 2t & 0 \\ -5z & 0 & 0 & -3px + 2t \end{pmatrix}$$

7.4 Traces and Contractions

7.4.1 Theoretical Review

The next operation one can perform with tensors is tensor **contraction**, which can be thought of as a generalization of the vector inner product. Contraction is performed by summing over a pair of indices, where each index appears once as a lower index and once as an upper index. An example of contraction that we have already seen is the contraction of a tensor with the metric, which raises or lowers an index. Similarly, the contraction of a tensor with a Jacobian or inverse Jacobian is a coordinate transformation.

Let us start with the simplest example we can, which should highlight the connection with the vector inner product, which can be seen as the contraction of a vector (having one upper index) and a covector (having one lower index):

$$v^a w_a = g_{ab} v^a w^b,$$

where the right-hand side of the equality comes from the fact that lowering the index on a vector w^b can be written as $w_a = g_{ab} w^b$, where g is the metric. Contractions involving higher rank tensors can then be seen as an extension of the inner product. For example:

$$M^{ab} N_{bc} = g_{bd} M^{ab} N^d{}_c.$$

Furthermore, we can also perform multiple contractions at the same time:

$$M^{ab} N_{ab} = g_{ac} g_{bd} M^{ab} N^{cd}.$$

We may even perform a contraction involving two indices on the same tensor, which is also known as taking the **trace**:

$$M^a{}_a = g_{ab} M^{ab}.$$

Finally, it is also perfectly valid to perform contractions on pairs of indices from more than two tensors at the same time, however such contractions may be broken down into individual operations. As an example, consider the following:

$$M^{ab}N_{bc}L^{cd} = g_{bi}g_{cj}M^{ab}N^{ij}L^{cd}.$$

The above contraction can be separated into individual contractions giving:

$$M^{ab}N_{bc}L^{cd} = M^{ab}(N_{bc}L^{cd}).$$

It is important to note that in a contraction, there are two different types of indices: the **contracted indices** which are summed over, and **free indices** which are not summed over. The result of performing a tensor contraction will be given by the number of free indices, so in the example above, the resulting tensor would have rank 2 as a and d are both free indices, while b and c are contracted indices.

7.4.2 PyOGRe Syntax

Contractions are performed in PyOGRe by supplying a tensor formula of the form `tensor1("indices1") @ tensor2("indices2")`, where `tensor1` and `tensor2` are the tensor objects that are to be contracted, and `"indices1"` and `"indices2"` are the index strings for each tensor. Any **repeated index** in either index string will be contracted.

This means that $v^a w_a$ can be calculated using `v("a") @ w("a")`, or $M^{ab}N_{bc}L^{cd}$ can be calculated using `M("a b") @ N("b c") @ L("c d")`. Notice that we do not specify whether each index is either an upper index or a lower index. This is because PyOGRe will automatically determine whether each index needs to be an upper index or lower index automatically, which can prevent one of the most common errors when performing these calculations by hand. The only important details is whether the index appears once or twice, and in what order they appear.

Let us create the stress-energy tensor for a perfect fluid with a 4-velocity u as a first example, which is given by:

$$T^{\mu\nu} = (\rho + p)u^\mu u^\nu + p\eta^{\mu\nu}$$

Notice that the above expression does not actually contain any contractions. This will demonstrate how we can use contractions as an **outer product**. This example will not only include contractions, but we will also be combining all previously seen operations such as addition and multiplication by a scalar, which PyOGRe will handle for us automatically.

[ ]:
```
perfect_fluid_from_velocity = og.Calc(
    (rho + p) * four_velocity("mu") @ four_velocity("nu") + p * minkowski("mu␣
 ↪nu"),
    name="Perfect fluid",
    symbol="T",
    indices="mu nu"
)
perfect_fluid_from_velocity.show()
```

Perfect fluid:

$$T_{\mu\nu}\left(t,x,y,z\right)=\begin{pmatrix}\frac{-pv^2-\rho}{v^2-1} & \frac{v(p+\rho)}{v^2-1} & 0 & 0\\ \frac{v(p+\rho)}{v^2-1} & \frac{-p-\rho v^2}{v^2-1} & 0 & 0\\ 0 & 0 & p & 0\\ 0 & 0 & 0 & p\end{pmatrix}$$

Substituting $v=0$ we also recover the stress-energy tensor we defined earlier:

[ ]:
```
perfect_fluid_from_velocity.show(indices=(1,1), replace={v: 0})
```

Perfect fluid:

$$T^{\mu\nu}\left(t,x,y,z\right)=\begin{pmatrix}\rho & 0 & 0 & 0\\ 0 & p & 0 & 0\\ 0 & 0 & p & 0\\ 0 & 0 & 0 & p\end{pmatrix}$$

We can also use the contraction syntax to multiply a tensor by a rank 0 tensor. Let us perform a contraction of the Kretschmann scalar with the Schwarzschild metric, which will give the same result as we saw for scalar multiplication:

[ ]:
```
og.Calc(
    kretschmann() @ schwarzschild("mu nu")
).show()
```

Result:

$$\Box_{\mu\nu}\left(t,r,\theta,\phi\right)=\begin{pmatrix}\frac{48M^2\cdot(2M-r)}{r^7} & 0 & 0 & 0\\ 0 & \frac{48M^2}{r^5(-2M+r)} & 0 & 0\\ 0 & 0 & \frac{48M^2}{r^4} & 0\\ 0 & 0 & 0 & \frac{48M^2\sin^2\left(\theta\right)}{r^4}\end{pmatrix}$$

Calculating the trace of a tensor may be done simply by supplying a matching pair of indices in that tensor's index string. As an example, the trace of the Minkowski metric can be calculated as follows:

[ ]:
```
og.Calc(
    minkowski("mu mu")
).show()
```

Result:

$$\square\,(t, x, y, z) = 4$$

And similarly for the trace of the stress-energy tensor of a perfect fluid:

[ ]:
```
og.Calc(
    perfect_fluid_from_velocity("mu mu")
).show()
```

Result:

$$\square\,(t, x, y, z) = 3p - \rho$$

PyOGRe also has a built-in helper function which can generate an index string using the same syntax as SymPy's `symbols()` function:

[ ]:
```
display(
    og.str_symbols("x0:10")
)
```

```
'x0 x1 x2 x3 x4 x5 x6 x7 x8 x9'
```

The output of `str_symbols()` may be supplied directly as an index string, which may be helpful in some scenarios. As an example, we can calculate the norm squared (the contraction of a tensor with itself in all indices) of the stress-energy tensor for a perfect fluid as follows using the `str_symbols()` function:

[ ]:
```
og.Calc(
    perfect_fluid_from_velocity(og.str_symbols("x0:2")) @␣
 ↪perfect_fluid_from_velocity(og.str_symbols("x0:2"))
).show()
```

Result:

$$\square\,(t, x, y, z) = 3p^2 + \rho^2$$

7.5 Norm Squared

A shorthand method for calculating the norm squared of any tensor is provided by simply calling the `calc_norm_squared()` method. The result is always a scalar and is simply the contraction of a tensor with itself in all indices. For a rank 2 tensor, that would be:

$$|T|^2 = T^a{}_b T_a{}^b.$$

For the Minkowski metric, we can calculate the norm squared as follows:

[ ]:
```
minkowski.calc_norm_squared().show()
```

Minkowski Norm Squared:

$$\square\,(t, x, y, z) = 4$$

Which for any metric, simply gives the number of dimensions in that metric. As another example, we can calculate the norm squared of the 4-velocity vector to ensure it has a norm squared of -1:

[ ]:
```
four_velocity.calc_norm_squared().show()
```

Four Velocity Norm Squared:

$$\square\,(t, x, y, z) = -1$$

7.6 Derivatives and Christoffel Symbols

7.6.1 Partial Derivative

The **partial derivative** ∂_μ can be used by importing the `PartialD` function from PyOGRe. This function can be contracted with other tensors using the same contraction syntax we saw earlier. We can calculate both gradients and divergences by supplying an appropriate index string.

[ ]:
```
from PyOGRe import PartialD
print(PartialD.__doc__)
```

```
    (function) PartialD(index)

    Represents the partial derivative when used within Calc()

    `index`: The index of the partial derivative used in tensor formulae.
```

The **gradient** of a tensor is the partial derivative acting with a free index on a tensor, which results in a tensor with a rank one higher than the tensor being acted on. As an example, we can calculate the gradient of the Kretschmann scalar as follows:

[ ]:
```
og.Calc(
    PartialD("mu") @ kretschmann()
).show(coords=spherical)
```

Result:

$$\square_\mu\,(t, r, \theta, \phi) = \begin{pmatrix} 0 \\ -\frac{288M^2}{r^7} \\ 0 \\ 0 \end{pmatrix}$$

The **divergence** of a tensor is instead the contraction of the partial derivative with one of the tensor's indices, resulting in a tensor of one rank lower than the tensor being acted on. Consider calculating the divergence of the Schwarzschild metric as an example:

[ ]:
```
og.Calc(
    PartialD("a") @ schwarzschild("a b")
).list()
```

Result:

No Non-Zero Elements

As we can see, the syntax for both the gradient and divergence is the same. Instead, if the index specification for the partial derivative matches one of the indices of the tensor, we get the divergence, otherwise if the index does not match, we get the gradient.

When working with the partial derivative, one should be mindful that in general, the result of applying the partial derivative to a tensor does not result in another tensor. This means that the result will not transform in the same way as a tensor does under a change of coordinates. It is for this reason that when working in general relativity, the **covariant derivative** is used instead of the partial derivative.

The partial derivative finds use in the definition of the covariant derivative itself, the Levi-Cevita connection, and the Riemann tensor, which will soon be discussed. Of these cases, both the covariant derivative and the Riemann tensor transform like tensors, while the Levi-Cevita connection whose components are known as the Christoffel symbols do not, and must be treated with special transformation rules. The partial derivative then should be avoided when doing calculations with the `Calc()` function unless you are familiar with these caveats.

7.6.2 Christoffel Symbols

As mentioned above, the **Christoffel symbols** are a class of very important "tensor-like" objects in differential geometry. They are the components of the **Levi-Civita connection**, which is the **unique** torsion-free connection that preserves the metric. They are "tensor-like" due to the fact that they do not transform as a tensor does, and must be transformed using a special set of rules under coordinate transformations.

The Christoffel symbols are defined as follows:

$$\Gamma^{\lambda}{}_{\mu\nu} = \frac{1}{2} g^{\lambda\sigma} \left(\partial_\mu g_{\nu\sigma} + \partial_\nu g_{\sigma\mu} - \partial_\sigma g_{\mu\nu} \right).$$

Each term in the parentheses is a gradient of the metric, each with a slightly different index configuration. We can calculate the Christoffel symbols manually in PyOGRe as follows:

[ ]:
```
og.Calc(
    (1/2) * schwarzschild("lambda sigma") @ (
        PartialD("mu") @ schwarzschild("nu sigma") +
        PartialD("nu") @ schwarzschild("sigma mu") -
        PartialD("sigma") @ schwarzschild("mu nu")
    ),
```

```
    name="Schwarzschild Christoffel Manual",
    symbol="Gamma",
).change_default_indices(indices=(1, -1, -1)).list()
```

Schwarzschild Christoffel Manual:

$$\Gamma^{t}{}_{tr} = \Gamma^{t}{}_{rt} = \frac{M}{r\left(-2M + r\right)}$$

$$\Gamma^{r}{}_{tt} = \frac{M\left(-2M + r\right)}{r^{3}}$$

$$\Gamma^{r}{}_{rr} = \frac{M}{r\left(2M - r\right)}$$

$$\Gamma^{r}{}_{\theta\theta} = 2M - r$$

$$\Gamma^{r}{}_{\phi\phi} = \left(2M - r\right)\sin^{2}\left(\theta\right)$$

$$\Gamma^{\theta}{}_{r\theta} = \Gamma^{\theta}{}_{\theta r} = \Gamma^{\phi}{}_{r\phi} = \Gamma^{\phi}{}_{\phi r} = \frac{1}{r}$$

$$\Gamma^{\theta}{}_{\phi\phi} = -\frac{\sin\left(2\theta\right)}{2}$$

$$\Gamma^{\phi}{}_{\theta\phi} = \Gamma^{\phi}{}_{\phi\theta} = \frac{1}{\tan\left(\theta\right)}$$

There is an easier way to calculate the Christoffel symbols in PyOGRe, which can simply be done by calling the `calc_christoffel()` method on any metric. Comparing the manual calculation above with the shorthand method below shows that the two calculations are equivalent.

[ ]:
```
schwarzschild_cs = schwarzschild.calc_christoffel()
schwarzschild_cs.list()
```

Schwarzschild Christoffel:

$$\Gamma^{t}{}_{tr} = \Gamma^{t}{}_{rt} = \frac{M}{r\left(-2M + r\right)}$$

$$\Gamma^{r}{}_{tt} = \frac{M\left(-2M + r\right)}{r^3}$$

$$\Gamma^{r}{}_{rr} = \frac{M}{r\left(2M - r\right)}$$

$$\Gamma^{r}{}_{\theta\theta} = 2M - r$$

$$\Gamma^{r}{}_{\phi\phi} = \left(2M - r\right)\sin^2\left(\theta\right)$$

$$\Gamma^{\theta}{}_{r\theta} = \Gamma^{\theta}{}_{\theta r} = \Gamma^{\phi}{}_{r\phi} = \Gamma^{\phi}{}_{\phi r} = \frac{1}{r}$$

$$\Gamma^{\theta}{}_{\phi\phi} = -\frac{\sin\left(2\theta\right)}{2}$$

$$\Gamma^{\phi}{}_{\theta\phi} = \Gamma^{\phi}{}_{\phi\theta} = \frac{1}{\tan\left(\theta\right)}$$

We do have an issue though. We mentioned earlier that the Christoffel symbols are not true tensors as they do not transform like tensors. To demonstrate this, let us first define a simple metric:

[ ]:
```
simple_metric = cartesian.new_metric(
    name="Simple Metric",
    components=sym.Array(
        [
            [-x, 0, 0, 0],
            [0, 1, 0, 0],
            [0, 0, 1, 0],
            [0, 0, 0, 1]
        ]
    ),
)
simple_metric.show()
```

Simple Metric:

$$g_{\mu\nu}\left(t, x, y, z\right) = \begin{pmatrix} -x & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Then we can calculate the Christoffel symbols using the built-in method `calc_christoffel()`. Having PyOGRe display the components in both Cartesian and Spherical coordinates gives us the following:

[ ]:
```
christoffel_auto = simple_metric.calc_christoffel()
christoffel_auto.list()
christoffel_auto.list(coords=spherical)
```

Simple Metric Christoffel:

$$\Gamma^{t}{}_{tx} = \Gamma^{t}{}_{xt} = \frac{1}{2x}$$

$$\Gamma^{x}{}_{tt} = \frac{1}{2}$$

Simple Metric Christoffel:

$$\Gamma^{t}{}_{tr} = \Gamma^{t}{}_{rt} = \frac{1}{2r}$$

$$\Gamma^{t}{}_{t\theta} = \Gamma^{t}{}_{\theta t} = \frac{1}{2\tan\left(\theta\right)}$$

$$\Gamma^{t}{}_{t\phi} = \Gamma^{t}{}_{\phi t} = -\frac{\tan\left(\phi\right)}{2}$$

$$\Gamma^{r}{}_{tt} = \frac{\sin\left(\theta\right)\cos\left(\phi\right)}{2}$$

$$\Gamma^{r}{}_{\theta\theta} = -r$$

$$\Gamma^{r}{}_{\phi\phi} = -r\sin^{2}\left(\theta\right)$$

$$\Gamma^{\theta}{}_{tt} = \frac{\cos\left(\phi\right)\cos\left(\theta\right)}{2r}$$

$$\Gamma^{\theta}{}_{r\theta} = \Gamma^{\theta}{}_{\theta r} = \Gamma^{\phi}{}_{r\phi} = \Gamma^{\phi}{}_{\phi r} = \frac{1}{r}$$

$$\Gamma^{\theta}{}_{\phi\phi} = -\frac{\sin\left(2\theta\right)}{2}$$

$$\Gamma^{\phi}{}_{tt} = -\frac{\sin\left(\phi\right)}{2r\sin\left(\theta\right)}$$

$$\Gamma^{\phi}{}_{\theta\phi} = \Gamma^{\phi}{}_{\phi\theta} = \frac{1}{\tan\left(\theta\right)}$$

Let us compare the above result with a manual calculation. First we can calculate the Christoffel symbols manually as we did for the Schwarzschild metric, and the request that PyOGRe display the components in both Cartesian and Spherical coordinates.

[ ]:
```
christoffel_manual = og.Calc(
    (1/2) * simple_metric("lambda sigma") @ (
        PartialD("mu") @ simple_metric("nu sigma") +
        PartialD("nu") @ simple_metric("sigma mu") -
        PartialD("sigma") @ simple_metric("mu nu")
    ),
    name="Simple Metric Christoffel Manual",
    symbol="Gamma"
)
christoffel_manual.change_default_indices(indices=(1,-1,-1))
christoffel_manual.list()
christoffel_manual.list(coords=spherical)
```

Simple Metric Christoffel Manual:

$$\Gamma^{t}{}_{tx} = \Gamma^{t}{}_{xt} = \frac{1}{2x}$$

$$\Gamma^{x}{}_{tt} = \frac{1}{2}$$

Simple Metric Christoffel Manual:

$$\Gamma^{t}{}_{tr} = \Gamma^{t}{}_{rt} = \frac{1}{2r}$$

$$\Gamma^{t}{}_{t\theta} = \Gamma^{t}{}_{\theta t} = \frac{1}{2\tan(\theta)}$$

$$\Gamma^{t}{}_{t\phi} = \Gamma^{t}{}_{\phi t} = -\frac{\tan(\phi)}{2}$$

$$\Gamma^{r}{}_{tt} = \frac{\sin(\theta)\cos(\phi)}{2}$$

$$\Gamma^{\theta}{}_{tt} = \frac{\cos(\phi)\cos(\theta)}{2r}$$

$$\Gamma^{\phi}{}_{tt} = -\frac{\sin(\phi)}{2r\sin(\theta)}$$

As you can see, calculating the Christoffel symbols in Cartesian coordinates manually agrees with the result of PyOGRe's calculation. However, when transforming the components of the manually calculated Christoffel symbols, we get the wrong answer. This is because PyOGRe does not know that the manually calculated Christoffel symbols do not transform like a tensor.

For this reason, when calculating the Christoffel symbols, we should **always** use the built-in method `calc_christoffel()` as PyOGRe will automatically flag the Christoffel symbols as a special tensor object. This then allows PyOGRe to apply the proper transformation rules.

As another example, let us first define the **Friedmann–Lemaitre–Robertson–Walker (FLRW) metric**, which describes an expanding universe:

[ ]:
```
a = sym.Function("a")(t)
k = sym.symbols("k")
flrw = spherical.new_metric(
    name="FLRW",
    components=sym.Array(
        [
            [-1, 0, 0, 0],
            [0, a**2/(1-k*r**2), 0, 0],
            [0, 0, a**2*r**2, 0],
            [0, 0, 0, a**2*r**2*sym.sin(theta)**2]
        ]
    )
)
flrw.show()
```

FLRW:

$$g_{\mu\nu}\left(t, r, \theta, \phi\right) = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & \frac{a^2(t)}{-kr^2+1} & 0 & 0 \\ 0 & 0 & r^2a^2(t) & 0 \\ 0 & 0 & 0 & r^2a^2(t)\sin^2\left(\theta\right) \end{pmatrix}$$

The function $a(t)$ is the **scale factor** and the parameter **k** is the curvature of the spatial surfaces. That is, $k = -1$ corresponds to positive curvature, $k = 0$ corresponds to a flat universe, and $k = 1$ corresponds to negative curvature. The Christoffel symbols are then easily calculated by calling the `calc_christoffel()` method on the metric:

[ ]:
```
flrw.calc_christoffel().list()
```

FLRW Christoffel:

$$\Gamma^{t}{}_{rr} = -\frac{a(t)\frac{d}{dt}a(t)}{kr^2-1}$$

$$\Gamma^{t}{}_{\theta\theta} = r^2 a(t)\frac{d}{dt}a(t)$$

$$\Gamma^{t}{}_{\phi\phi} = r^2 a(t)\sin^2(\theta)\frac{d}{dt}a(t)$$

$$\Gamma^{r}{}_{tr} = \Gamma^{r}{}_{rt} = \Gamma^{\theta}{}_{t\theta} = \Gamma^{\theta}{}_{\theta t} = \Gamma^{\phi}{}_{t\phi} = \Gamma^{\phi}{}_{\phi t} = \frac{\frac{d}{dt}a(t)}{a(t)}$$

$$\Gamma^{r}{}_{rr} = -\frac{kr}{kr^2-1}$$

$$\Gamma^{r}{}_{\theta\theta} = kr^3 - r$$

$$\Gamma^{r}{}_{\phi\phi} = r\left(kr^2-1\right)\sin^2(\theta)$$

$$\Gamma^{\theta}{}_{r\theta} = \Gamma^{\theta}{}_{\theta r} = \Gamma^{\phi}{}_{r\phi} = \Gamma^{\phi}{}_{\phi r} = \frac{1}{r}$$

$$\Gamma^{\theta}{}_{\phi\phi} = -\frac{\sin(2\theta)}{2}$$

$$\Gamma^{\phi}{}_{\theta\phi} = \Gamma^{\phi}{}_{\phi\theta} = \frac{1}{\tan(\theta)}$$

7.6.3 Covariant Derivative

Since the partial derivative does not transform like a tensor, it is less than ideal for use in general relativity, apart from it being used in a handful of cases such as the Christoffel symbols. The **covariant derivative** ∇_μ on the other hand, does in-fact transform like a tensor and is simply a generalization of the partial derivative. The covariant derivative can be defined as follows:

- For a scalar s, the covariant derivative is simply the partial derivative $\nabla_\mu s = \partial_\mu s$.
- For a vector v^ν, the covariant derivative is given by $\nabla_\mu v^\nu = \partial_\mu v^\nu + \Gamma^{\nu}{}_{\mu\lambda} v^\lambda$.
- For a covector w_ν, the covariant derivative is given by $\nabla_\mu v_\nu = \partial_\mu w_\nu - \Gamma^{\lambda}{}_{\mu\nu} w_\lambda$.

The above definitions can be generalized to work on a rank (p, q) tensor $T^{a_1\ldots a_p}_{b_1\ldots b_q}$, giving the covariant derivative $\nabla_\mu T^{a_1\ldots a_p}_{b_1\ldots b_q}$ as:

- The partial derivative $\partial_\mu T^{a_1\ldots a_p}_{b_1\ldots b_q}$.
- **Adding** one term $\Gamma^{a_i}{}_{\mu\lambda} T^{a_1\ldots\lambda\ldots a_p}_{b_1\ldots b_q}$ for each upper index a_i.
- And **subtracting** one term $\Gamma^{\lambda}{}_{\mu b_i} T^{a_1\ldots a_p}_{b_1\ldots\lambda\ldots b_q}$ for each lower index b_i.

Even though the above definition includes the partial derivative, it turns out that the covariant derivative does actually transform like a tensor. Under a change of coordinates, the unwanted terms from the partial derivative exactly cancel the additional unwanted terms from the Christoffel symbols.

Using PyOGRe, we can manually calculate the covariant derivative just like any other tensor formula. As an example, we can calculate the covariant divergence of the Schwarzschild metric as follows:

[ ]:
```
og.Calc(
    PartialD("mu") @ schwarzschild("alpha beta") -
    schwarzschild_cs("lambda mu alpha") @ schwarzschild("lambda beta") -
    schwarzschild_cs("lambda mu beta") @ schwarzschild("alpha lambda"),
).list()
```

Result:

No Non-Zero Elements

However, PyOGRe provides a simpler way of calculating the covariant derivative using `CovariantD()`, which will automatically add in the required terms as detailed above for each of the tensor's indices. Using the `CovariantD()` function in a tensor formula is identical to the partial derivative.

[ ]:
```
from PyOGRe import CovariantD
print(CovariantD.__doc__)
```

```
    (function) CovariantD(index)

    Represents the covariant derivative when used within Calc()

    `index`: The index of the covariant derivative used in tensor formulae.
```

As an example, let us re-calculate the covariant divergence of the Schwarzschild metric using `CovariantD()`:

[ ]:
```
og.Calc(
    CovariantD("mu") @ schwarzschild("alpha beta")
).list()
```

Result:

No Non-Zero Elements

As we can see, the covariant divergence is identically zero for the Schwarschild metric, which should be the case, since as we noted previously, the Levi-Cevita connection preserves the metric. As further proof to this claim, we can calculate the covariant divergence of the FLRW metric and we see that it also vanishes:

[ ]:
```
og.Calc(
    CovariantD("mu") @ flrw("alpha beta")
).list()
```

Result:

No Non-Zero Elements

7.7 Curvature Tensors

7.7.1 Riemann Tensor

The **Riemann curvature tensor** $R^{\rho}{}_{\sigma\mu\nu}$ can be calculated from the Christoffel symbols given the following definition:

$$R^{\rho}{}_{\sigma\mu\nu} = \partial_{\mu}\Gamma^{\rho}{}_{\nu\sigma} - \partial_{\nu}\Gamma^{\rho}{}_{\mu\sigma} + \Gamma^{\rho}{}_{\mu\lambda}\Gamma^{\lambda}{}_{\nu\sigma} - \Gamma^{\rho}{}_{\nu\lambda}\Gamma^{\lambda}{}_{\mu\sigma}.$$

Again, just like the covariant derivative, the Riemann tensor still transforms like a tensor does under a change of coordinates since the unwanted terms from the partial derivatives and Christoffel symbols cancel each other. PyOGRe provides a shorthand method `calc_riemann_tensor()` which can be called on any metric tensor to calculate the Riemann tensor using the above definition.

As an example, let us calculate the Riemann tensor for the FLRW metric:

[ ]:
```
flrw.calc_riemann_tensor().list()
```

FLRW Riemann:

$$R^{t}{}_{rtr} = -R^{t}{}_{rrt} = -\frac{a(t)\frac{d^2}{dt^2}a(t)}{kr^2 - 1}$$

$$R^{t}{}_{\theta t\theta} = -R^{t}{}_{\theta\theta t} = r^2 a(t)\frac{d^2}{dt^2}a(t)$$

$$R^{t}{}_{\phi t\phi} = -R^{t}{}_{\phi\phi t} = r^2 a(t)\sin^2(\theta)\frac{d^2}{dt^2}a(t)$$

$$R^{r}{}_{ttr} = -R^{r}{}_{trt} = R^{\theta}{}_{tt\theta} = -R^{\theta}{}_{t\theta t} = R^{\phi}{}_{tt\phi} = -R^{\phi}{}_{t\phi t} = \frac{\frac{d^2}{dt^2}a(t)}{a(t)}$$

$$R^{r}{}_{\theta r\theta} = R^{\phi}{}_{\theta\phi\theta} = r^2\left(k + \left(\frac{d}{dt}a(t)\right)^2\right)$$

$$R^{r}{}_{\theta\theta r} = R^{\phi}{}_{\theta\theta\phi} = r^2\left(-k - \left(\frac{d}{dt}a(t)\right)^2\right)$$

$$R^{r}{}_{\phi r\phi} = R^{\theta}{}_{\phi\theta\phi} = r^2\left(k + \left(\frac{d}{dt}a(t)\right)^2\right)\sin^2(\theta)$$

$$R^{r}{}_{\phi\phi r} = R^{\theta}{}_{\phi\phi\theta} = r^2\left(-k - \left(\frac{d}{dt}a(t)\right)^2\right)\sin^2(\theta)$$

$$R^{\theta}{}_{rr\theta} = R^{\phi}{}_{rr\phi} = \frac{k + \left(\frac{d}{dt}a(t)\right)^2}{kr^2 - 1}$$

$$R^{\theta}{}_{r\theta r} = R^{\phi}{}_{r\phi r} = \frac{-k - \left(\frac{d}{dt}a(t)\right)^2}{kr^2 - 1}$$

Using the built-in method for calculating the Riemann tensor not only makes the calculation simpler, but it can also be more performant. Whenever PyOGRe calculates any of the pre-defined curvature tensors, PyOGRe will also calculate and store any of the intermediate steps for later use. If any of the prerequisite curvature tensors have already been calculated, then PyOGRe will **not** re-perform the calculation and will instead retrieve the previous result. For this reason, it is recommended to use the built-in methods for calculating any of the curvature tensors instead of calculating them manually.

As a final example, we can calculate the **Kretschmann scalar** directly from the Schwarzschild metric, which is simply the norm squared of the Riemann tensor:

[ ]:
```
schwarzschild.calc_riemann_tensor().calc_norm_squared().show()
```

Schwarzschild Riemann Norm Squared:

$$\Box\left(t, r, \theta, \phi\right) = \frac{48M^2}{r^6}$$

7.7.2 Ricci Tensor

The **Ricci tensor** $R_{\mu\nu}$ is the trace of the first and third indices of the Riemann tensor:

$$R_{\mu\nu} = R^{\lambda}{}_{\mu\lambda\nu}.$$

The shorthand method `calc_ricci_tensor()` can be called on any metric tensor to calculate the Ricci tensor using the above definition, just as we did for the Riemann tensor. Let us calculate the Ricci tensor for the FLRW metric as an example:

[ ]:
```
flrw.calc_ricci_tensor().list()
```

FLRW Ricci Tensor:

$$R_{tt} = -\frac{3\frac{d^2}{dt^2}a(t)}{a(t)}$$

$$R_{rr} = \frac{-2k - a(t)\frac{d^2}{dt^2}a(t) - 2\left(\frac{d}{dt}a(t)\right)^2}{kr^2 - 1}$$

$$R_{\theta\theta} = r^2 \cdot \left(2k + a(t)\frac{d^2}{dt^2}a(t) + 2\left(\frac{d}{dt}a(t)\right)^2\right)$$

$$R_{\phi\phi} = r^2 \cdot \left(2k + a(t)\frac{d^2}{dt^2}a(t) + 2\left(\frac{d}{dt}a(t)\right)^2\right)\sin^2\left(\theta\right)$$

7.7.3 Ricci Scalar

In a similar vain, the **Ricci scalar** R is the trace of the Ricci tensor:

$$R = R^{\lambda}{}_{\lambda}.$$

The Ricci scalar can be calculated in PyOGRe by simply calling the `calc_ricci_scalar()` method on any metric tensor. Continuing our examples with the FLRW metric, let us calculate the Ricci scalar:

[ ]:
```
flrw.calc_ricci_scalar().list()
```

FLRW Ricci Scalar:

$$R = \frac{6\left(k + a(t)\frac{d^2}{dt^2}a(t) + \left(\frac{d}{dt}a(t)\right)^2\right)}{a^2(t)}$$

7.7.4 Einstein Tensor

Moving on, the **Einstein tensor** $G_{\mu\nu}$ can be calculated using the following definition:

$$G_{\mu\nu} = R_{\mu\nu} - \frac{1}{2}g_{\mu\nu}R.$$

As with all the other curvature tensors, PyOGRe provides a shortcut method of calculating the Einstein tensor by calling the `calc_einstein_tensor()` method on any metric tensor. Let us calculate the Einstein tensor for the FLRW metric as an example:

[ ]:
```
flrw.calc_einstein_tensor().list()
```

FLRW Einstein:

$$G_{tt} = \frac{3\left(k + \left(\frac{d}{dt}a(t)\right)^2\right)}{a^2(t)}$$

$$G_{rr} = \frac{k + 2a(t)\frac{d^2}{dt^2}a(t) + \left(\frac{d}{dt}a(t)\right)^2}{kr^2 - 1}$$

$$G_{\theta\theta} = r^2\left(-k - 2a(t)\frac{d^2}{dt^2}a(t) - \left(\frac{d}{dt}a(t)\right)^2\right)$$

$$G_{\phi\phi} = r^2\left(-k - 2a(t)\frac{d^2}{dt^2}a(t) - \left(\frac{d}{dt}a(t)\right)^2\right)\sin^2(\theta)$$

7.7.5 Weyl Tensor

The **Weyl tensor** $C_{\rho\sigma\mu\nu}$ can be calculated using the definition:

$$C_{\rho\sigma\mu\nu} = R_{\rho\sigma\mu\nu} - \frac{2}{n-2}(R_{\rho\nu}g_{\sigma\mu} - R_{\rho\mu}g_{\sigma\nu} + R_{\sigma\mu}g_{\rho\nu} - R_{\sigma\nu}g_{\rho\mu}) + \frac{1}{(n-1)(n-2)}(R(g_{\rho\mu}g_{\sigma m} - g_{\rho\nu}g_{\sigma\mu})).$$

where n is the number of dimensions. Note that the Weyl tensor is only defined in three or more dimensions, but otherwise it is very similar to the Riemann tensor. In PyOGRe, the Weyl tensor can be calculated by using the `calc_weyl_tensor()` method on any metric tensor. Let us calculate the Weyl tensor for the FLRW metric to continue our examples:

[ ]:
```
ws = schwarzschild.calc_weyl_tensor()
ws.list()
```

Schwarzschild Weyl Tensor:

$$C_{trtr} = -C_{trrt} = -C_{rttr} = C_{rtrt} = -\frac{2M}{r^3}$$

$$C_{t\theta t\theta} = C_{\theta t\theta t} = \frac{M\left(-2M + r\right)}{r^2}$$

$$C_{t\theta\theta t} = C_{\theta tt\theta} = \frac{M\left(2M - r\right)}{r^2}$$

$$C_{t\phi t\phi} = C_{\phi t\phi t} = \frac{M\left(-2M + r\right)\sin^2\left(\theta\right)}{r^2}$$

$$C_{t\phi\phi t} = C_{\phi tt\phi} = \frac{M\left(2M - r\right)\sin^2\left(\theta\right)}{r^2}$$

$$C_{r\theta r\theta} = C_{\theta r\theta r} = \frac{M}{2M - r}$$

$$C_{r\theta\theta r} = C_{\theta rr\theta} = \frac{M}{-2M + r}$$

$$C_{r\phi r\phi} = C_{\phi r\phi r} = \frac{M\sin^2\left(\theta\right)}{2M - r}$$

$$C_{r\phi\phi r} = C_{\phi rr\phi} = \frac{M\sin^2\left(\theta\right)}{-2M + r}$$

$$C_{\theta\phi\theta\phi} = -C_{\theta\phi\phi\theta} = -C_{\phi\theta\theta\phi} = C_{\phi\theta\phi\theta} = 2Mr\sin^2\left(\theta\right)$$

The Weyl tensor is designed so that any possible contraction vanishes, which we can verify for the first two indices as follows:

[ ]:
```
og.Calc(
    ws("a a mu nu")
).list()
```

Result:

No Non-Zero Elements

7.7.6 Non-trivial Covariant Derivative Example

Before moving on to the next section, let us provide a non-trivial example of the covariant derivative by calculating the **energy-momentum conservation equations** for the FLRW metric. First, let us consider the covariant divergence of the Einstein tensor which is given by:

$$\nabla_\mu G^{\mu\nu} = \partial_\mu G^{\mu\nu} + \Gamma^\mu{}_{\mu\lambda} G^{\lambda\nu} + \Gamma^\nu{}_{\mu\lambda} G^{\mu\lambda} = 0.$$

The divergence of the Einstein tensor vanishes due to the **Bianchi identity**:

$$\nabla_\mu R^{\mu\nu} - \frac{1}{2}\nabla^\nu R \to \nabla_\mu G^{\mu\nu} = 0.$$

In PyOGRe, this can be calculated as follows:

[ ]:
```
flrw_einstein = flrw.calc_einstein_tensor()
og.Calc(
    CovariantD("mu") @ flrw_einstein("mu nu")
).list()
```

Result:

No Non-Zero Elements

Next, we note that the stress-energy tensor should be **conserved**, that is:

$$\nabla_\mu T^{\mu\nu} = \partial_\mu T^{\mu\nu} + \Gamma^\mu{}_{\mu\lambda} T^{\lambda\nu} + \Gamma^\nu{}_{\mu\lambda} T^{\mu\lambda} = 0.$$

This follows from the fact that the divergence of the Einstein tensor vanishes, together with the **Einstein equation**:

$$G_{\mu\nu} = \alpha T_{\mu\nu},$$

where the value of α is dependent on the system of units (we will work with $\alpha = 1$). In order to derive the energy-momentum conservation equations for the FLRW metric, we must first define the stress-energy tensor. To start, we can define the rest-frame 4-velocity u^μ:

[ ]:
```
flrw_rest_velocity = flrw.new_tensor(
    name="FLRW Rest Velocity",
    components=sym.Array(
        [1, 0, 0, 0]
    ),
    indices=(1,),
    symbol="u"
)
flrw_rest_velocity.show()
```

FLRW Rest Velocity:

$$u^\mu\left(t, r, \theta, \phi\right) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Then using the four velocity, we can define the stress-energy tensor for a perfect fluid using the formula:

$$T^{\mu\nu} = (\rho + p) u^\mu u^\nu + p g^{\mu\nu},$$

where both ρ and p can be given a coordinate dependence.

[ ]:
```
f_rho = sym.Function("rho")(t, r, theta, phi)
f_p = sym.Function("p")(t, r, theta, phi)

flrw_perfect_fluid = og.Calc(
    (f_rho + f_p) * flrw_rest_velocity("mu") @ flrw_rest_velocity("nu") + f_p *␣
 ↪flrw("mu nu"),
    name="FLRW Perfect Fluid",
    symbol="T"
)
flrw_perfect_fluid.show()
```

FLRW Perfect Fluid:

$$T^{\mu\nu}\left(t,r,\theta,\phi\right) = \begin{pmatrix} \rho(t,r,\theta,\phi) & 0 & 0 & 0 \\ 0 & \frac{\left(-kr^2+1\right)p(t,r,\theta,\phi)}{a^2(t)} & 0 & 0 \\ 0 & 0 & \frac{p(t,r,\theta,\phi)}{r^2a^2(t)} & 0 \\ 0 & 0 & 0 & \frac{p(t,r,\theta,\phi)}{r^2a^2(t)\sin^2(\theta)} \end{pmatrix}$$

Finally, we can take the covariant divergence of the stress-energy tensor:

[ ]:
```
flrw_conservation = og.Calc(
    CovariantD("mu") @ flrw_perfect_fluid("mu nu"),
    name="FLRW Conservation",
    symbol="C"
)
flrw_conservation.list()
```

FLRW Conservation:

$$C^t = \frac{3\left(p(t,r,\theta,\phi)+\rho(t,r,\theta,\phi)\right)\frac{d}{dt}a(t) + a(t)\frac{\partial}{\partial t}\rho(t,r,\theta,\phi)}{a(t)}$$

$$C^r = \frac{\left(-kr^2+1\right)\frac{\partial}{\partial r}p(t,r,\theta,\phi)}{a^2(t)}$$

$$C^\theta = \frac{\frac{\partial}{\partial\theta}p(t,r,\theta,\phi)}{r^2a^2(t)}$$

$$C^\phi = \frac{\frac{\partial}{\partial\phi}p(t,r,\theta,\phi)}{r^2a^2(t)\sin^2(\theta)}$$

By requiring the t component vanishes, we find the following equation:

$$\dot{\rho} = -3(\rho + p)\frac{\dot{a}}{a}$$

That is, in an expanding universe, energy is **not** conserved, but instead the energy density changes in a way that is dependent on the scale factor. Should the universe not be expanding, the $\dot{a} = 0$ and energy is once again conserved.

8 Curves and Geodesics

The next major feature of PyOGRe is the ability to calculate geodesic equations. The geodesic equations describe the **worldline** of a particle that is free from all external forces, and they are dependent on the geometry of the spacetime.

8.1 Setting the Curve Parameter

Before we start calculating geodesics, it may be useful to choose a **curve parameter**. By default, PyOGRe will use λ to parameterize the geodesic equations, but this may be changed using the `set_curve_parameter()` function.

[ ]:
```
print(og.set_curve_parameter.__doc__)
```

```
    (function) set_curve_parameter(symbol)

    This function will either print the existing curve parameter, or will
overwrite it if an argument is given.

    `symbol`: The symbol to set as the curve parameter. Supplying an argument of
'automatic' will set the index letters to the default.
```

To see the current curve parameter, we can call the function with no argument:

[ ]:
```
og.set_curve_parameter()
```

λ

To change the parameter, we can call the function with a string which will be used to define the curve parameter, or a SymPy symbol:

[ ]:
```
og.set_curve_parameter("kappa")
```

κ

[ ]:
```
og.set_curve_parameter(sym.symbols("tau"))
```

τ

To reset the curve parameter to the default value λ, we can call the function with "automatic" as the argument:

[ ]:
```
og.set_curve_parameter("automatic")
```

λ

8.2 The Curve Lagrangian

Let us assume we are given a **curve** that is a function $x^\mu(\lambda)$ in the given spacetime, where λ is the curve parameter. Then, the **curve Lagrangian** of the metric is defined to be the norm squared of the tangent to said curve:

$$L = g^{\mu\nu}\dot{x}^\mu\dot{x}^\nu,$$

where $\dot{x}^\mu$ is the first derivative of x^μ with respect to the curve parameter using Newtownian dot notation. In PyOGRe, the curve lagrangian can be calculated using the `calc_lagrangian()` method on a metric. As an example, below is the curve lagrangian of the Minkowski metric:

[ ]:
```
minkowski.calc_lagrangian().show()
```

Minkowski Lagrangian:

$$L\left(t,x,y,z\right) = -\dot{t}^2 + \dot{x}^2 + \dot{y}^2 + \dot{z}^2$$

Just like any other object in PyOGRe, we can have PyOGRe display the curve lagrangian in any coordinate system as long as the transformation rules are supplied:

[ ]:
```
minkowski.calc_lagrangian().show(coords=spherical)
minkowski.calc_lagrangian(coords=spherical).show()
```

Minkowski Lagrangian:

$$L\left(t,r,\theta,\phi\right) = \dot{\phi}^2 r^2\sin^2\left(\theta\right) + r^2\dot{\theta}^2 + \dot{r}^2 - \dot{t}^2$$

Minkowski Lagrangian:

$$L\left(t,r,\theta,\phi\right) = \dot{\phi}^2 r^2\sin^2\left(\theta\right) + r^2\dot{\theta}^2 + \dot{r}^2 - \dot{t}^2$$

As a final example, we can calculate the curve lagrangian of both the Schwarzschild metric and the FLRW metric:

[ ]:
```
schwarzschild.calc_lagrangian().list()
flrw.calc_lagrangian().show()
```

Schwarzschild Lagrangian:

$$L = \frac{r^3\cdot\left(2M-r\right)\left(\dot{\phi}^2\sin^2\left(\theta\right)+\dot{\theta}^2\right) - r^2\dot{r}^2 + \dot{t}^2\left(2M-r\right)^2}{r\left(2M-r\right)}$$

FLRW Lagrangian:

$$L\left(t, r, \theta, \phi\right) = \frac{-\dot{r}^2 a^2(t) + \left(kr^2 - 1\right)\left(\dot{\phi}^2 r^2 a^2(t) \sin^2\left(\theta\right) + r^2 \dot{\theta}^2 a^2(t) - \dot{t}^2\right)}{kr^2 - 1}$$

8.3 Geodesics From the Lagrangian

Once we have the curve lagrangian, we can calculate the geodesic equations from it using the **Euler-Lagrange** equations:

$$\frac{\partial L}{\partial x^\mu} - \frac{\mathrm{d}}{\mathrm{d}\lambda}\left(\frac{\partial L}{\partial \dot{x}^\mu}\right) = 0$$

In PyOGRe, we can calculate the geodesic equations from the Euler-Lagrange equations by calling the `calc_geodesic_from_lagrangian()` method on a metric. So, for the Minkowski metric, we get:

[ ]:
```
minkowski.calc_geodesic_from_lagrangian().list()
```

Minkowski Geodesic From Lagrangian:

$$0^t = -\ddot{t}$$

$$0^x = \ddot{x}$$

$$0^y = \ddot{y}$$

$$0^z = \ddot{z}$$

Notice that the coordinate dependence on the curve parameter is not included when using the `show()` or `list()` methods. If instead we requested the components as a SymPy array, the curve parameter dependence would become clear:

[ ]:
```
minkowski.calc_geodesic_from_lagrangian().get_components()
```

[ ]:
$$\left[-\ddot{t}(\lambda) \quad \ddot{x}(\lambda) \quad \ddot{y}(\lambda) \quad \ddot{z}(\lambda)\right]$$

Next, for the Schwarzschild metric, we get:

[ ]:
```
schwarzschild.calc_geodesic_from_lagrangian().list(replace={M: 1, theta: 0, phi:
↪ 0})
```

Schwarzschild Geodesic From Lagrangian:

$$0^t = -\ddot{t} + \frac{2\ddot{t}}{r} - \frac{2\dot{r}\dot{t}}{r^2}$$

$$0^r = r^4\ddot{r} - 2r^3\ddot{r} - r^2\dot{r}^2 + r^2\dot{t}^2 - 4r\dot{t}^2 + 4\dot{t}^2$$

Notice that we were also able to supply substitutions to the geodesic equations. When performing substitutions in the geodesic equations, PyOGRe will automatically perform the substitutions for any derivatives as well. In the above example, since we substituted $\theta = 0$ and $\phi = 0$, PyOGRe automatically knew to substitute $\dot{\theta} = \ddot{\theta} = \dot{\phi} = \ddot{\phi} = 0$ as well.

As a final example, the geodesic equations for the FLRW metric with $k = 0$, $\theta = 0$, and $\phi = 0$ are given by:

[ ]:
```
flrw.calc_geodesic_from_lagrangian().list(replace={k: 0, theta: 0, phi: 0})
```

FLRW Geodesic From Lagrangian:

$$0^t = \dot{r}^2 a(t) \frac{d}{dt} a(t) + \ddot{t}$$

$$0^r = \left(\ddot{r} a(t) + 2\dot{r}\dot{t} \frac{d}{dt} a(t) \right) a(t)$$

8.4 Geodesics From the Christoffel Symbols

An alternative method of obtaining the geodesic equations can be done using the covariant derivative and the Christoffel symbols. With this method, the geodesic equations are given by:

$$\dot{x}^\mu \nabla_\mu \dot{x}^\nu = 0 \rightarrow \ddot{x}^\mu + \Gamma^\mu{}_{\rho\sigma} \dot{x}^\rho \dot{x}^\sigma = 0.$$

In PyOGRe, the above equation can be calculated using the `calc_geodesic_from_christoffel()` method on a metric. For example:

[ ]:
```
minkowski.calc_geodesic_from_christoffel().list()
schwarzschild.calc_geodesic_from_christoffel().list(replace={M: 1, theta: 0,␣
↪phi: 0})
flrw.calc_geodesic_from_christoffel().list(replace={k: 0, theta:0, phi: 0})
```

Minkowski Geodesic From Christoffel:

$$0^t = \ddot{t}$$

$$0^x = \ddot{x}$$

$$0^y = \ddot{y}$$

$$0^z = \ddot{z}$$

Schwarzschild Geodesic From Christoffel:

$$0^t = r^2 \ddot{t} - 2r\ddot{t} + 2\dot{r}\dot{t}$$

$$0^r = r^4 \ddot{r} - 2r^3 \ddot{r} - r^2 \dot{r}^2 + r^2 \dot{t}^2 - 4r\dot{t}^2 + 4\dot{t}^2$$

FLRW Geodesic From Christoffel:

$$0^t = -\dot{r}^2 a(t) \frac{d}{dt} a(t) - \ddot{t}$$

$$0^r = -\ddot{r} a(t) - 2\dot{r}\dot{t} \frac{d}{dt} a(t)$$

Both methods of determining the geodesic equations can be useful; in some cases one method will yield simpler equations that can more easily be solved than the other. The best thing one can do is to simply try both methods and see which one gives nicer and/or simpler equations for the specific metric in question.

8.5 Geodesics in Terms of the Time Coordinate

Finally, if we have a spacetime metric, it may be useful to instead parameterize the geodesic equations in terms of the time coordinate. It can be shown that the geodesic equations would be given by the following expression when put in terms of the time coordinate:

$$\frac{\mathrm{d}^2 x^\mu}{\mathrm{d}t^2} + \left(\Gamma^\mu{}_{\rho\sigma} - \Gamma^0{}_{\rho\sigma} \frac{\mathrm{d}x^\mu}{\mathrm{d}t} \right) \frac{\mathrm{d}x^\rho}{\mathrm{d}t} \frac{\mathrm{d}x^\sigma}{\mathrm{d}t} = 0.$$

In PyOGRe, these equations can be obtained by using the `calc_geodesic_with_time_parameter()` method on a metric. For example:

[ ]:
```
minkowski.calc_geodesic_with_time_parameter().list()
schwarzschild.calc_geodesic_with_time_parameter().list(replace={M: 1, theta: 0,␣
↪phi: 0})
flrw.calc_geodesic_with_time_parameter().list(replace={k: 0, theta: 0, phi: 0})
```

Minkowski Geodesic With Time Parameter:

$$0^x = \ddot{x}$$

$$0^y = \ddot{y}$$

$$0^z = \ddot{z}$$

Schwarzschild Geodesic With Time Parameter:

$$0^r = r^4 \ddot{r} - 2r^3 \ddot{r} - 3r^2 \dot{r}^2 + r^2 - 4r + 4$$

FLRW Geodesic With Time Parameter:

$$0^r = -\ddot{r} a(t) + \dot{r}^3 a^2(t) \frac{d}{dt} a(t) - 2\dot{r} \frac{d}{dt} a(t)$$

Notice that we only need a maximum of three equations instead of four when parameterizing the geodesic equations in terms of the time coordinate.

9 Additional Information

9.1 Version History

The full version history and change log is available on the GitHub repository, in the CHANGELOG.md file.

9.2 Feature Requests and Bug Reports

The package is under continuous development. If you have any feature requests or bug reports, please feel free to open a new issue on GitHub. If you would like to provide any new functionality, please feel free to submit a pull request.

9.3 Copyright and Citing

No citing information available at the moment.