OGRe$^{\text{TM}}$: An **O**bject-oriented **G**eneral **Re**lativity **T**oolkit for **M**athematica

By Barak Shoshany (baraksh@gmail.com) (baraksh.com)

Documentation for v1.0 (February 10, 2021)

https://github.com/bshoshany/OGRe

Introduction

Summary

OGRe is a modern Mathematica package for tensor calculus, designed to be both powerful and user-friendly. It can be used in a variety of contexts where tensor calculations are needed, in both mathematics and physics, but it is especially suitable for general relativity.

Tensors are abstract objects, which can be represented as multi-dimensional arrays once a choice of index configuration and coordinate system is made. OGRe stays true to this definition, but takes away the complexities that come with combining tensors in different representations. This is done using an object-oriented programming approach, as detailed in the documentation.

The user initially defines each tensor in OGRe using its explicit components in any single representation. Operations on this tensor are then done abstractly, without needing to specify which representation to use. Possible operations include addition of tensors, multiplication of tensor by scalar, trace, contraction, and partial and covariant derivatives.

OGRe will automatically choose which representation to use for each tensor based on how the tensors are combined. For example, if two tensors are added, then OGRe will automatically use the same index configuration for both. Similarly, if two tensors are contracted, then OGRe will automatically ensure that the contracted indices are one upper (contravariant) and one lower (covariant). OGRe will also automatically transform all tensors being operated on to the same coordinate system.

Transformations between representations are done behind the scenes; all the user has to do is specify which metric to use for raising and lowering indices, and how to transform between the coordinate systems being used. This also means that there is no room for user error. The user cannot mistakenly perform "illegal" operations such as $2\,A^{\mu\nu} + B_{\mu\lambda}\,C_{\lambda\nu}$. Instead, the user simply inputs the names of the tensors, the order (but **not** the configuration) of indices for each, and the operations to perform - and the correct combination $2\,A^{\mu\nu}\,B^{\mu}{}_{\lambda}\,C^{\lambda\nu}$ will be automatically deduced.

I initially created OGRe for use in my own research, so I made it as flexible and powerful as possible. I also wanted my students to be able to use it easily and efficiently, even if they only have minimal experience with Mathematica and/or general relativity, so I made it simple to learn and easy to use. As a result, this package is equally suitable for both experienced and novice researchers.

Features

- Define coordinate systems and the transformation rules between them. Tensor components are then transformed automatically between coordinates behind the scenes as needed.
- Each tensor is associated with a specific metric. Tensor components are then transformed automatically between different index configurations, raising and lowering indices behind the scenes as needed.
- Display any tensor in any index configuration and coordinate system, either in vector/matrix form or as a list of all unique non-zero elements.
- Automatically simplify tensor components, optionally with user-defined simplification assumptions.
- Export tensors to a Mathematica notebook or to a file, so they can later be imported into another Mathematica session without having to redefine them from scratch.
- Easily calculate arbitrary tensor formulas using any combination of addition, multiplication by scalar, trace, contraction, partial derivative, and covariant derivative.
- Built-in modules for calculating the Christoffel symbols (Levi-Civita connection), Riemann tensor, Ricci tensor and scalar, and Einstein tensor. More will be added in future versions.
- Fully portable. Can be imported directly from the web into any Mathematica notebook, without downloading or installing anything.
- Clear and detailed documentation, with many examples, in both Mathematica notebook and PDF format. Detailed usage messages are also provided.
- Open source. The code is extensively documented; please feel free to fork and modify it as you see fit.
- Under active development. Please see the "future plans" section of the documentation for more information. Bug reports and feature requests are welcome, and should be made via GitHub issues.

The object-oriented design philosophy

Object-oriented programming refers to a paradigm where a program’s code is organized around objects. An **object** belongs to a user-defined type, called a **class**. The class defines the **data** that the object stores, as well as **methods** or **member functions** that read or manipulate that data. One of the fundamental principles of object-oriented programming is **encapsulation**, which means that the user may only access an object’s data using the methods defined by the class, and is unable to access the object’s data directly.

Importantly, encapsulation allows for the preservation of **class invariants**. An invariant is a condition of validity that can always be assumed to be satisfied by the data stored in each object. If the methods make sure to preserve the invariant whenever they store or manipulate the data, and the user is prevented from changing the data manually and thus potentially violating the invariant, then the implementation of the class can be greatly simplified, and performance can be improved, because we will not need to verify that the data is valid every time we perform an operation.

The main idea behind OGRe is to simplify the use of tensors by encoding all the information about a tensor in a single, self-contained object. As we mentioned above, a tensor is an abstract object. We can find components which represent this abstract entity in a particular coordinate system and index configuration, but the tensor is **not** its components. In OGRe, a tensor object is initially defined (or **constructed**) by providing the components of the tensor in a particular representation - but once this is done, the user does not need to worry about coordinates or indices anymore, or even remember which coordinates and indices were initially used. The abstract tensor object will automatically transform the initial data to a different coordinate system or index configuration as needed, based on the context in which it was used.

As a tensor object holds the components of the same tensor in many different representations, the most important class invariant is the assumption that the different components indeed represent the same tensor. This is achieved using encapsulation; the object’s data can only be modified by private methods that preserve the invariant, and thus the user cannot accidentally cause a violation of the invariant by assigning components to one representation that are not related to the components of all other representations by the appropriate coordinate and/or index transformation.

Unfortunately, Mathematica does not have built-in support for object-oriented programming. However, version 10.0 of Mathematica, released in 2014, introduced the `Association` symbol. An `Association` is an **associative array**; it is similar to a `List`, except that instead of being just an array of values, an `Association` is a list of keys with a value associated to each key. This allows us to easily implement a rudimentary form of object-oriented programming, storing the properties of each object in the keys of a corresponding `Association`.

Of course, as Mathematica is not truly object-oriented, there is no actual “tensor class” defined anywhere in the package. Instead, the tensor class exists only **implicitly**, as a design paradigm. Furthermore, the functions that process the data stored in the tensor objects are not methods of a class, they are simply modules that take tensor objects as input and/or produce tensor objects as outputs. (In earlier versions, I tried using a syntax that resembled method syntax in languages such as C++ or Python, but eventually decided against it, as it was too cumbersome.) Still, designing the OGRe toolkit with object-oriented programming in mind allows us to reap many of this paradigm’s benefits, as explained above - and it simply makes sense for tensors, due to their abstract and multifaceted nature.

Installing and loading the package

The OGRe toolkit consists of only one file, `OGRe.m`. There are several different ways to load the package:

- **Run from local file with installation:** This is the recommended option, as it allows you to permanently use the package offline from any Mathematica notebook. Download the file `OGRe.m` from https://github.com/bshoshany/OGRe and copy it to the directory given by `FileNameJoin[{$UserBaseDirectory,"Applications"}]`. The package may now be loaded from any notebook by writing `Needs["OGRe`"]` (note the backtick ` following the word `OGRe`).

- **Run from local file without installation:** This option allows you to use the package in a portable fashion, without installing it in the `Applications` directory. Download the file `OGRe.m` from https://github.com/bshoshany/OGRe, place it in the same directory as the notebook you would like to use, and use the command `Get["OGRe.m",Path→NotebookDirectory[]]` to load the package.
- **Run from web with installation:** This option allows you to automatically download and install the package on any computer. Simply run the command `URLDownload["https://raw.githubusercontent.com/bshoshany/OGRe/master/OGRe.m",FileNameJoin[{$UserBaseDirectory,"Applications","OGRe.m"}]]` from any Mathematica notebook to permanently install the package. Then use `Needs["OGRe`"]` from any notebook to load it.
- **Run from web without installation:** This option allows you to use the package from any Mathematica notebook on any computer, without manually downloading or installing it, as long as you have a working Internet connection. Simply write `Get["https://raw.githubusercontent.com/bshoshany/OGRe/master/OGRe.m"]` in any Mathematica notebook to load the package.

To **uninstall** the package, just delete the file from the `Applications` directory, which can be done from within Mathematica using the command `DeleteFile[FileNameJoin[{$UserBaseDirectory,"Applications","OGRe.m"}]]`.

For the purposes of this documentation, I will use the "run from local file without installation" option, since you most likely downloaded both the documentation and the package together:

```
In[∘]:= Get["OGRe.m", Path → NotebookDirectory[]]
```

```
OGRe™: An (O)bject-oriented (G)eneral (Re)lativity (T)oolkit for (M)athematica
By Barak Shoshany (baraksh@gmail.com) (baraksh.com)
v1.0 (February 10, 2021)
To download the latest version, visit https://github.com/bshoshany/OGRe.
To list all available modules, type ?OGRe`*.
To get help on a particular module, type ? followed by the module name.
```

The package displays a welcome message upon loading, which provides some information on how to get started. As stated in the welcome message, to list all of the modules available in this package, you may use the following command:

In[ ]:=
```
? OGRe`*
```

Out[ ]=
```
OGRe`
TAddCoordTransformation   TCovariantD     TIndexLetters        TRicciScalar
TCalc                     TDelete         TInitializeSymbols   TRicciTensor
TChangeDefaultCoords      TEinsteinTensor TList                TRiemannTensor
TChangeDefaultIndices     TExport         TNewCoordinates      TShow
TChangeID                 TExportAll      TNewMetric           TSimplify
TChangeSymbol             TImport         TNewTensor           TSimplifyAssumptions
TChristoffel              TImportAll      TPartialD
```

Clicking on the name of any module in this list will show its usage message. Notice that all OGRe modules start with the letter `T`, to help distinguish them from other modules.

Creating and displaying tensor objects

Defining coordinates

To define tensors, we first need to define the manifold on which they reside. Since we are focusing on general relativity, we will use 4-dimensional spacetime manifolds in the following examples, but this toolkit works equally well with manifolds that are purely spatial and/or have a different number of dimensions.

The first step is to define the coordinate system. In OGRe, coordinates are represented as a special tensor object: a vector x^μ (a tensor of rank 1) representing a point. To define the coordinates, we use the module **`TNewCoordinates`**:

In[ ]:=
```
? TNewCoordinates
```

Out[ ]=

Symbol

TNewCoordinates[*coordinatesID, symbols*] creates a new tensor object representing a coordinate system.

coordinatesID is a string that will be used to identify the new object, and must be unique. If a tensor object with the same ID already exists, it will be overwritten.

symbols are the coordinate symbols, e.g. {t, x, y, z}. They will automatically be cleared and protected against future changes using TInitializeSymbols[ ].

For example, to define the **Cartesian coordinate system**, we use the following syntax:

```
In[ ]:= TNewCoordinates["Cartesian", {t, x, y, z}]
Out[ ]= Cartesian
```

The first argument is the new tensor object's **unique ID**. This is the string that we will use to refer to this tensor from now on, and it will also be displayed whenever we print the tensor. The ID string is case-sensitive, can include any characters, and can be of any length, but it is recommended to keep it short and simple. If we create another tensor object with the same ID, the previous definition will be overwritten. Note that the ID string is also the return value of the module; generally, all modules that operate on a tensor object will return its ID string as output. This allows us to compose different modules together, as we will see below.

The second argument is the list of coordinates. Note that the order of coordinates matters, as it will determine the order of components in the tensors defined on this manifold. The symbols used also matter, as tensor components will usually be functions of these symbols. We can similarly define the spherical coordinate system:

```
In[ ]:= TNewCoordinates["Spherical", {t, r, θ, ϕ}]
Out[ ]= Spherical
```

It is crucial that the coordinate symbols, in this case `t`, `r`, `θ`, `ϕ`, remain as **undefined** symbols throughout the calculation, otherwise errors may occur. For example, if our metric contains functions of `r`, and at some point in the notebook we set `r = 1`, then Mathematica will replace every instance of `r` with `1`, which means those abstract functions will be replaced with their values evaluated at `r = 1`. Furthermore, if we, for example, want to take the derivative with respect to `r` (e.g. for the purpose of calculating various curvature tensors), this will not be possible, since one cannot take a derivative with respect to a number.

To prevent such errors, `TNewCoordinates` automatically clears any previous definitions of the given symbols, and also protects them against future changes. This is done using the `TInitializeSymbols` module:

```
In[ ]:= ? TInitializeSymbols
```

```
Out[ ]= Symbol

TInitializeSymbols[symbol1, symbol2, …] clears any definitions previously used for the given symbols and prot⋱
    ects them against future changes.
Useful for making sure coordinate variables, constants, and abstract functions used in tensors do not acc⋱
    identally change their definitions and break the code.
```

Indeed, if we now try to give `r` the value `1`, we will get an error:

```
In[ ]:= r = 1

Set: Symbol r is Protected.

Out[ ]= 1
```

You can also use `TInitializeSymbols` manually for any constants or parameters used in the tensor, as we will demonstrate in the next section. Note that it is always possible to replace protected symbols with numbers or other expressions using `/.` or `ReplaceAll`.

Defining metrics

To finish defining a manifold, we need to define its metric tensor. For this, we use the module `TNewMetric`:

```
In[ ]:= ? TNewMetric
```

```
Out[ ]= Symbol

TNewMetric[metricID, coordinatesID, components, symbol] creates a new tensor object representing a metric.
metricID is a string that will be used to identify the new object, and must be unique. If a tensor object with th⋱
    e same ID already exists, it will be overwritten.
coordinatesID is the unique ID of a tensor object representing a coordinate system, created using TNewC⋱
    oordinates[ ].
components is a square matrix representing the metric with two lower indices in that coordinate system.
symbol will be used to represent the metric in formulas. If not given, "g" will be used.
```

Again, we must input a unique ID for the new tensor object corresponding to the metric. We also input the unique ID of a coordinate system created using `TNewCoordinates`. This coordinate system is the one in which the components of the metric are initially given, but they will be automatically transformed to other coordinate systems later as needed. Note that the components are assumed to be the representation of the metric with two lower indices, since that is how metrics are usually defined; one upper and one lower index is just the identity matrix, and two upper indices is the inverse metric. Optionally, we can also specify a symbol to be used for representing the metric.

Let us use this module to create a tensor object for the **Minkowski metric**, specifying the components in Cartesian coordinates:

```
In[ ]:= TNewMetric["Minkowski", "Cartesian", DiagonalMatrix[{-1, 1, 1, 1}], "η"]
Out[ ]= Minkowski
```

As with `TNewCoordinates`, we received the ID string for the new tensor object as output.

Similarly, let us define the **Schwarzschild metric**, this time specifying the components in spherical coordinates. However, before we can safely do so, we should take one additional step. When we defined the coordinates using `TNewCoordinates`, the symbols used were automatically initialized, that is, cleared and protected from future changes, using `TInitializeSymbols`. However, `TNewMetric` does not automatically initialize symbols. Since the Schwarzschild metric has a free parameter `M`, the mass of the black hole, we must initialize this symbol manually:

```
In[ ]:= TInitializeSymbols[M]
```

Now we can define the Schwarzschild metric:

```
In[ ]:= TNewMetric["Schwarzschild", "Spherical",
          DiagonalMatrix[{-(1 - 2 M/r), 1/(1 - 2 M/r), r^2, r^2 Sin[θ]^2}]]
Out[ ]= Schwarzschild
```

Note that since we do not specify a symbol, the symbol **“g”** will be used by default, as demonstrated below.

Displaying tensors

OGRe contains two modules for displaying the contents of tensors. The first one is `TShow`, which shows the ID, symbol, indices, coordinates, and components in those indices and coordinates, in vector or matrix form when applicable:

In[∘]:=
```
? TShow
```

Out[∘]=

Symbol

TShow[*ID*, *indices*, *coordinatesID*] shows the components of the tensor object *ID* with the index configuration *indices* and in the coordinate system *coordinatesID*, in vector or matrix form when applicable.

indices should be a list of the form {±1, ±1, ...}, where +1 corresponds to an upper index and −1 corresponds to a lower index.

If the index configuration and/or coordinate system are omitted, the default ones will be used.

Coordinates are also tensor objects, so we can use `TShow` to show the two coordinate tensors we defined above:

In[∘]:=
```
TShow["Cartesian"]
```

Out[∘]=

$$\text{Cartesian:}\quad x^{\mu} = \begin{pmatrix} t \\ x \\ y \\ z \end{pmatrix}$$

In[∘]:=
```
TShow["Spherical"]
```

Out[∘]=

$$\text{Spherical:}\quad x^{\mu} = \begin{pmatrix} t \\ r \\ \theta \\ \phi \end{pmatrix}$$

Note that coordinate tensors always have the symbol *x*.

We can also show the two metrics we created using these coordinates:

In[∘]:=
```
TShow["Minkowski"]
```

Out[∘]=

$$\text{Minkowski:}\quad \eta_{\mu\nu}(t, x, y, z) = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In[∘]:=
```
TShow["Schwarzschild"]
```

Out[∘]=

$$\text{Schwarzschild:}\quad g_{\mu\nu}(t,r,\theta,\phi)=\begin{pmatrix}-1+\frac{2M}{r} & 0 & 0 & 0\\ 0 & \frac{1}{1-\frac{2M}{r}} & 0 & 0\\ 0 & 0 & r^2 & 0\\ 0 & 0 & 0 & r^2\,\text{Sin}[\theta]^2\end{pmatrix}$$

The other module available in OGRe for displaying the contents of tensors is `TList`, which lists all of the **unique (up to sign), non-zero** components of the tensor. It is usually the best option for higher-rank tensors, which cannot be displayed in vector or matrix form, such as the Christoffel symbols or Riemann tensor (see below). Its syntax is:

In[∘]:=
```
? TList
```

Out[∘]=

Symbol

TList[*ID, indices, coordinatesID*] lists the unique, non–zero components of the tensor object *ID* with the index c⋱

onfiguration *indices* and in the coordinate system *coordinatesID*.

indices should be a list of the form {±1, ±1, ...}, where +1 corresponds to an upper index and −1 correspon⋱

ds to a lower index.

If the index configuration and/or coordinate system are omitted, the default ones will be used.

For example:

In[∘]:=
```
TList["Minkowski"]
```

Out[∘]=

Minkowski:

$$\eta_{tt}=-\eta_{xx}=-\eta_{yy}=-\eta_{zz}\;=\;-1$$

In[∘]:=
```
TList["Schwarzschild"]
```

Out[∘]=

Schwarzschild:

$$g_{tt} = -1+\frac{2M}{r}$$
$$g_{rr} = \frac{1}{1-\frac{2M}{r}}$$
$$g_{\theta\theta} = r^2$$
$$g_{\phi\phi} = r^2\,\mathrm{Sin}[\theta]^2$$

Note that both `TShow` and `TList` display their outputs using the `DisplayFormula` Notebook style. It's up to the user to decide how to define this style; in this notebook, I used a font size of 20 and aligned to center. The style may be easily changed by clicking on the “Format” menu in Mathematica and selecting ”Edit Stylesheet”. Then, choose the `DisplayFormula` style, select that cell, and modify its format using the “Format” menu.

If, as in the examples above, no additional arguments are given to `TShow` and `TList`, they display the tensors in their **default indices** and **default coordinates**, which are the ones first used to define the tensor (unless you change them later). So, for example, the default indices of the Minkowski metric are two lower indices, and its default coordinates are Cartesian. We will show later how to change these defaults, and how to display any tensor in any index configuration and coordinate system.

By default, the `TShow` module uses Greek letters for the indices, in a specific order. The letters can be displayed or changed using the `TIndexLetters` module:

In[∘]:=
```
? TIndexLetters
```

Out[∘]=

Symbol

TIndexLetters[*letters*] changes the letters to be used when displaying indices.

If *letters* is omitted, shows the index letters currently in use.

If *letters* is set to Automatic, resets the index letters to the default.

The default letters are:

In[∘]:=
```
TIndexLetters[]
```

Out[∘]=
```
μνρσκλαβγδεζϵθιξπτϕχψω
```

This means that the letter μ will be used for the first index, ν for the second, and so on. However, sometimes we want to use different letters. For example, let us change the indices to lowercase English letters:

In[∘]:=
```
TIndexLetters["abcdefghijklmnopqrstuvwxyz"]
```

“Show” will now use these letters - in this particular order - when displaying tensors:

In[∘]:=
```
TShow["Minkowski"]
```

Out[∘]=

$$\text{Minkowski:}\quad \eta_{ab}(t, x, y, z) = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Finally, let us reset the letters to the default:

In[∘]:=
```
TIndexLetters[Automatic]
```

Note that `TList` always uses the coordinate symbols for the indices, so it is not affected by `TIndexLetters`.

Creating tensors in a given manifold

Any tensors other than coordinates and metrics are created using the module `TNewTensor`:

In[∘]:=
```
? TNewTensor
```

Out[∘]=

Symbol

TNewTensor[*tensorID, metricID, indices, components, symbol*] creates a new tensor object.

tensorID is a string that will be used to identify the new object, and must be unique. If a tensor object with the s⋱

ame ID already exists, it will be overwritten.

metricID is the unique ID of a tensor object representing a metric, created using TNewMetric[ ]. The metric w⋱

ill be used to raise and lower indices for the new tensor.

indices is a list of the form {±1, ±1, ...}, where +1 corresponds to an upper index and −1 corresponds to a low⋱

er index.

components is a list specifying the representation of the tensor, with the specified index configuration, in the d⋱

efault coordinate system of *metricID*.

symbol will be used to represent the tensor in formulas. If not given, the placeholder ⬚ will be used.

In OGRe, all tensor objects must have an associated metric - except coordinate objects, and the metric tensors themselves. This is because OGRe automatically raises and lowers indices as appropriate for various operations such as adding and contracting tensors, and it cannot do so without knowing which metric to use. Even scalars, which have no indices, should still be associated to a specific metric - since they can multiply other tensors, and you don't want to multiply two tensors from different manifolds.

The index configuration of the tensor is a 1-dimensional `List`. The number of indices is the rank of the tensor. Each element in the `List` corresponds to one index, with `+1` specifying an upper index and `-1` specifying a lower index. For example, `{-1,-1}` corresponds to a tensor such as the metric $g_{\mu\nu}$, which has two lower indices, while `{1,-1,-1,-1}` corresponds to a tensor such as the Riemann tensor $R^{\rho}{}_{\sigma\mu\nu}$, which has one upper index followed by three lower indices.

The components of the tensor must also be a `List`. The components are the representation of the new tensor in the given index configuration and in the **default coordinate system of the metric** we are using to create it. The components will be automatically converted to different indices or coordinates later as needed, as we will demonstrate below.

To create a **scalar**, or a tensor of rank 0 (with no indices), we must input an empty list `{}` for the indices, and a list with one item for the components. For example, let us define the **Kretschmann scalar** in the Schwarzschild spacetime (below we will show how to calculate it directly from the metric):

In[∘]:=
```
TNewTensor["Kretschmann", "Schwarzschild", {}, {48 M^2/r^6}, "K"]
```
Out[∘]=
```
Kretschmann
```

Again, the output is the unique ID of the tensor object that was created. Let us show the tensor:

In[∘]:=
```
TShow["Kretschmann"]
```
Out[∘]=

$$\text{Kretschmann:}\quad \mathrm{K}\,(t, r, \theta, \phi) = \frac{48\,M^2}{r^6}$$

Notice that the output of `TNewTensor` is also the input of `TShow`, so in fact, we could compose them together using `@`. We will do so from now on.

Similarly, we can create a **vector**, or a tensor of rank 1 (with one index). For example, let us create a vector for the 4-velocity of a particle moving at 3-velocity *v* along the *x* direction in Minkowski space. First we initialize the parameter representing the velocity:

In[∘]:=
```
TInitializeSymbols[v]
```

Since the 4-velocity has an upper index by definition, we make sure to define the components in the representation of the tensor with an upper index by specifying the index configuration as `{1}`:

In[ ]:=
```
TShow@TNewTensor["FourVelocity", "Minkowski", {1}, {1, v, 0, 0}/Sqrt[1 - v^2]]
```

Out[ ]=

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

Again, the output of `TNewTensor` was the ID of the tensor, `"FourVelocity"`, but that is also the input we want to pass to `TShow`, so we composed the two modules together. Note also that since we did not specify a symbol for this tensor, its symbol is just a placeholder ⬚. We will give it a proper symbol below.

Finally, as an example of a tensor of rank 2 (with two indices), let us define the stress-energy tensor $T^{\mu\nu}$ for a perfect fluid. First, let us initialize the symbols ρ (for the energy density) and **p** (for the pressure):

In[ ]:=
```
TInitializeSymbols[ρ, p]
```

Next we create the tensor, using its matrix representation with two upper indices by specifying the index configuration `{1,1}`:

In[ ]:=
```
TShow@TNewTensor["PerfectFluid", "Minkowski", {1, 1}, DiagonalMatrix[{ρ, p, p, p}], "T"]
```

Out[ ]=

$$\text{PerfectFluid:}\quad \mathrm{T}^{\mu\nu}(t, x, y, z) = \begin{pmatrix} \rho & 0 & 0 & 0 \\ 0 & p & 0 & 0 \\ 0 & 0 & p & 0 \\ 0 & 0 & 0 & p \end{pmatrix}$$

In a similar manner, we could also define tensors of rank 3 and above. However, such tensors are most often derived by operating on lower-rank tensors, rather than defined manually via their components. We will see an example of such a derivation when we derive the Christoffel symbols and Riemann tensor from the metric below.

Operations on single tensors

Changing the symbol or ID of a tensor

If we ever want to change the symbol used to display a tensor, we can simply use the module `TChangeSymbol`:

In[∘]:=
```
? TChangeSymbol
```

Out[∘]=
```
Symbol
TChangeSymbol[ID, symbol] changes the symbol of the tensor object ID to symbol.
```

For example, let us give the symbol *u* to the four-velocity, and then show it:

In[∘]:=
```
TShow@TChangeSymbol["FourVelocity", "u"]
```

Out[∘]=

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

Similarly, we can also change the ID of the tensor using the **`TChangeID`** module:

In[∘]:=
```
? TChangeID
```

Out[∘]=
```
Symbol
TChangeID[oldID, newID] changes the ID of the tensor object oldID to newID.
If the tensor is a metric or a coordinate system, all currently defined tensors will be scanned, and any referenc⋱
    es to oldID will be replaced with newID. If a tensor with the ID newID already exists, it will be overwritten.
```

For example, let us change the ID of the 4-velocity tensor from **`“FourVelocity”`** to **`“4-Velocity”`**:

In[∘]:=
```
TChangeID["FourVelocity", "4-Velocity"];
```

The old ID no longer represents any tensor object, so we get an error if we try using it:

In[∘]:=
```
TShow["FourVelocity"]
```

```
OGRe`Private`CheckIfTensorExists: The tensor "FourVelocity" does not exist.
```

Out[∘]=
```
$Aborted
```

We can access the tensor using the new ID:

In[∘]:=
```
TShow["4-Velocity"]
```

Out[∘]=

$$\text{4–Velocity:}\quad u^{\mu}(t, x, y, z) = \begin{pmatrix} \frac{1}{\sqrt{1-v^2}} \\ \frac{v}{\sqrt{1-v^2}} \\ 0 \\ 0 \end{pmatrix}$$

Note that when we define a tensor using a metric and a coordinate system, OGRe doesn’t store the actual metric components or coordinates inside the tensor object - it only stores **references** to the relevant objects, using their IDs. This both improves performance and allows us to modify the metric or coordinates later without having to redefine all the tensors derived from them. For this reason, if the tensor to be renamed represents a metric or a coordinate system, OGRe will automatically update the references in the definitions of all of the tensors that have been defined so far in the session using that metric or coordinate system. This guarantees that there are never any broken references.

If we want to delete a tensor we have created, we can use the `TDelete` module:

In[∘]:=
```
? TDelete
```

Out[∘]=
```
Symbol

TDelete[ID] deletes the tensor object ID.

WARNING: The tensor data will be lost forever and cannot be recovered.

If the tensor is a metric or coordinate system, it cannot be deleted unless all tensors referring to it have been d⋱
      eleted first.
```

To prevent breaking references, `TDelete` will not delete a tensor object representing a metric or coordinate system if it is referred to by any other tensor. For example, if we try to delete the coordinate system **`“Cartesian”`**, we will get an error message, since it is used as the default coordinate system for **`“Minkowski”`** (among others):

In[∘]:=
```
TDelete["Cartesian"]
```

```
TDelete: The coordinate system "Cartesian" cannot be deleted, as it is the default coordinate system of the tensor "Minkowski". To
   delete the coordinate system, first change the default coordinate system of "Minkowski" and any other relevant tensors.
```

Out[∘]=
```
$Aborted
```

Similarly, we cannot delete the metric **`“Minkowski”`** since it was used to define the tensor **`“PerfectFluid”`** (among others):

```
In[]:= TDelete["Minkowski"]
```

```
TDelete: The metric "Minkowski" cannot be deleted, as it has been used to define the tensor "PerfectFluid". To delete the metric,
    first delete "PerfectFluid" and any other tensors defined using this metric.
```

```
Out[]= $Aborted
```

Finally, note that there is no module to change the components of a tensor after it has already been defined, as this may break class invariants. Instead, simply create a new tensor with the same ID using the `TNewTensor` module, which will overwrite the previously defined tensor.

Raising and lowering indices

Raising and lowering indices is one of the most basic tensor operations. For example, if we have a vector represented with one upper index, v^ν, we can turn it into a covector, which is represented with one lower index, by **contracting** it with the metric:

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

This is called “lowering an index”. Here and in the rest of this documentation, we will be using the **Einstein summation convention**, where the same index repeated **exactly twice**, once as an upper index and once as a lower index, implies summation over that index. In this case, the implied summation is over $\nu \in \{0, 1, 2, 3\}$:

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

Such a sum over an index is called a **contraction**, and it is a generalization of the inner product, as we will describe in more details below. Conversely, if we have a covector w_μ, we can raise its index by contracting it with the inverse metric:

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

This works the same for indices of higher-rank tensors. For example, if we have a tensor of rank 2 represented with two upper indices, $T^{\mu\lambda}$, we can lower either one or both of its indices:

$$T^\mu{}_\nu = g_{\nu\lambda} T^{\mu\lambda}, \quad T_{\mu\nu} = g_{\mu\rho} g_{\nu\lambda} T^{\rho\lambda}.$$

In OGRe, since tensor objects are **abstract tensors**, independent of any specific index configuration, **there is no notion of raising or lowering the indices of a tensor object**. Instead, one simply request to display the components of the tensor with the desired index configuration. This works with both the `TShow` and `TList` modules, by simply adding as a second argument the list of indices in the format `{±1,±1,...}`, as when we created a new tensor.

As an example, let us show the vector **”`4-Velocity`”** with a lower index, that is, with index configuration `{-1}`:

In[ ]:=
```
TShow["4-Velocity", {-1}]
```
Out[ ]=
$$\text{4–Velocity:}\quad u_\mu(t,x,y,z)=\begin{pmatrix}-\frac{1}{\sqrt{1-v^2}}\\ \frac{v}{\sqrt{1-v^2}}\\ 0\\ 0\end{pmatrix}$$

OGRe automatically knows to use the Minkowski metric to lower the index, which means that a minus sign has been added to the first component, as expected. Similarly, let us lower just the **second** index on `PerfectFluid`:

In[ ]:=
```
TList["PerfectFluid", {1, -1}]
```
Out[ ]=

PerfectFluid:

$$T^t{}_t = -\rho$$
$$T^x{}_x = T^y{}_y = T^z{}_z = p$$

The components of the representation of the metric with two upper indices are the components of the inverse metric, since

$$g_{\mu\lambda}\, g^{\lambda\nu} = \delta^\nu_\mu .$$

Therefore, a quick way to show the components of the inverse metric is:

In[ ]:=
```
TShow["Schwarzschild", {1, 1}]
```
Out[ ]=
$$\text{Schwarzschild:}\quad g^{\mu\nu}(t,r,\theta,\phi)=\begin{pmatrix}\frac{r}{2M-r} & 0 & 0 & 0\\ 0 & 1-\frac{2M}{r} & 0 & 0\\ 0 & 0 & \frac{1}{r^2} & 0\\ 0 & 0 & 0 & \frac{\mathrm{Csc}[\theta]^2}{r^2}\end{pmatrix}$$

For the same reason, the metric with one upper and one lower index is just the identity matrix:

In[ ]:=
```
TList["Schwarzschild", {1, -1}]
```
Out[ ]=

Schwarzschild:

$$g^t{}_t = g^r{}_r = g^\theta{}_\theta = g^\phi{}_\phi = 1$$

As explained above, if the modules `TShow` or `TList` are called without any arguments, the tensor is displayed in its **default index configuration**, which is the one first used to define the tensor. So the 4-velocity has one upper index by default, and the stress tensor has two upper indices by default, because that is how we initially defined them. However, the default indices can be changed using the module `TChangeDefaultIndices`:

```
In[∘]:= ? TChangeDefaultIndices
```

```
Out[∘]=
Symbol

TChangeDefaultIndices[ID, indices] changes the default index configuration of the tensor object ID to indices.

indices is a list of the form {±1, ±1, ...}, where +1 corresponds to an upper index and −1 corresponds to a lower i⁛
    ndex.
```

For example, let us change the default indices of the perfect fluid stress tensor to two lower indices:

```
In[∘]:= TChangeDefaultIndices["PerfectFluid", {-1, -1}];
```

Now, when we display the tensor using `TShow` without any arguments, this is the index configuration that will be used:

```
In[∘]:= TShow["PerfectFluid"]
```

Out[∘]=

$$\text{PerfectFluid:}\quad T_{\mu\nu}(t,x,y,z)=\begin{pmatrix} p & 0 & 0 & 0 \\ 0 & p & 0 & 0 \\ 0 & 0 & p & 0 \\ 0 & 0 & 0 & p \end{pmatrix}$$

Coordinate transformations

The components of any tensor may be transformed from one coordinate system x^{μ} to another coordinate system $x^{\mu'}$ using the following prescription:

- For every lower index μ, add a factor of $\partial x^{\mu}/\partial x^{\mu'}$ (i.e. the derivative of the old coordinates with respect to the new, or the **Jacobian**).
- For every upper index μ, add a factor of $\partial x^{\mu'}/\partial x^{\mu}$ (i.e. the derivative of the new coordinates with respect to the old, or the inverse of the Jacobian).

For example, given a tensor with components $T_{\alpha\beta}$ in a coordinate system x^{μ}, we can transform to components $T_{\alpha'\beta'}$ in another coordinate system $x^{\mu'}$ as follows:

$$T_{\alpha'\beta'}\left(x^{\mu'}\right) = \frac{\partial x^{\alpha}}{\partial x^{\alpha'}}\frac{\partial x^{\beta}}{\partial x^{\beta'}}T_{\alpha\beta}\left(x^{\mu}\right).$$

For a general rank (p, q) tensor with p upper indices $\alpha_1, \ldots, \alpha_p$ and q lower indices $\beta_1, \ldots, \beta_q$, the transformation takes the form

$$T^{\alpha'_1\cdots\alpha'_p}_{\beta'_1\cdots\beta'_q}\left(x^{\mu'}\right) = \left(\frac{\partial x^{\alpha'_1}}{\partial x^{\alpha_1}}\cdots\frac{\partial x^{\alpha'_p}}{\partial x^{\alpha_p}}\right)\left(\frac{\partial x^{\beta_1}}{\partial x^{\beta'_1}}\cdots\frac{\partial x^{\beta_q}}{\partial x^{\beta'_q}}\right)T^{\alpha_1\cdots\alpha_p}_{\beta_1\cdots\beta_q}\left(x^{\mu}\right).$$

As a mnemonic for this formula, recall that two indices may only be contracted if one of them is an upper index and the other is a lower index. If an index is in the denominator of a derivative, then its role is reversed (upper ↔ lower). Thus the old (non-primed) and new (primed) indices can only be in places that allow properly contracting the Jacobian or inverse Jacobian with the tensor. For example, α_1 is an upper index in T and therefore must be contracted with a lower index. Thus, ∂x^{α_1} must be in the denomi nator, to lower its index and allow it to be contracted with the tensor.

As we saw above, OGRe automatically knows how to raise or lower indices as needed using the appropriate metric. Similarly, any operation that requires transforming to another coordinate system will preform the transformation automatically behind the scenes. However, for this to happen, OGRe needs to know the appropriate **transformation rules**. These are defined between the tensor objects representing the coordinates, which were generated using the module `TNewCoordinates`. The rules for transforming from a source coordinate system to a target coordinate system are stored within the tensor object representing the source. This is done using the module `TAddCoordTransformation`:

In[∘]:=
```
? TAddCoordTransformation
```

Out[∘]=

Symbol

TAddCoordTransformation[*sourceID, targetID, rules*] adds a transformation from the coordinate system *sourceID* to the coordinate system *targetID*.

The argument *rules* must be a list of transformation rules. For example, {x → r Sin[θ] Cos[ϕ], y → r Sin[θ] Sin[ϕ], z → r Cos[θ]} is a transformation from Cartesian to spherical coordinates.

Let us add the rules to transform from Cartesian to spherical coordinates:

In[∘]:=
```
TAddCoordTransformation["Cartesian", "Spherical",
  {x → r Sin[θ] Cos[ϕ], y → r Sin[θ] Sin[ϕ], z → r Cos[θ]}];
```

These will be stored in the data of the object `”Cartesian”`. Note that we did not have to input a rule for `t`, since in this case, it stays the same. Conversely, let us add the rules to transform from spherical to Cartesian coordinates:

In[∘]:=
```
TAddCoordTransformation["Spherical", "Cartesian",
  {r → Sqrt[x^2 + y^2 + z^2], θ → ArcCos[z / Sqrt[x^2 + y^2 + z^2]], ϕ → ArcTan[x, y]}];
```

These will be stored in the data of the object **`"Spherical"`**. Now OGRe knows how to convert back and forth between these two coordinate systems - and this will happen automatically whenever required. We just needed to provide these rules once and for all, and any tensor initially defined in one coordinate system can now be automatically converted to the other.

As in the case of raising and lowering indices, displaying a tensor in a different coordinate system is a simple matter of calling the modules `TShow` or `TList` with an additional argument specifying the coordinate system to use. For example, let us show the Minkowski metric in spherical coordinates:

In[∘]:=
```
TShow["Minkowski", "Spherical"]
```

Out[∘]=

$$\text{Minkowski:}\quad \eta_{\mu\nu}(t,r,\theta,\phi)=\begin{pmatrix}-1&0&0&0\\0&1&0&0\\0&0&r^2&0\\0&0&0&r^2\,\mathrm{Sin}[\theta]^2\end{pmatrix}$$

We can also ask to see a tensor in a specific index configuration **and** a specific coordinate system:

In[∘]:=
```
TShow["PerfectFluid", {1, 1}, "Spherical"]
```

Out[∘]=

$$\text{PerfectFluid:}\quad T^{\mu\nu}(t,r,\theta,\phi)=\begin{pmatrix}\rho&0&0&0\\0&p&0&0\\0&0&\frac{p}{r^2}&0\\0&0&0&\frac{p\,\mathrm{Csc}[\theta]^2}{r^2}\end{pmatrix}$$

The module `TList` works in exactly the same way, for example:

In[∘]:=
```
TList["Kretschmann", "Cartesian"]
```

Out[∘]=

Kretschmann:

$$K \quad = \quad \frac{48\,M^2}{(x^2+y^2+z^2)^3}$$

Just as with default indices, every tensor has a **default coordinate system**, which is, initially, the one we used to create it. We can change it using the module `TChangeDefaultCoords`, and then whenever we display the tensor, it will be displayed in that coordinate system if no other coordinate system is specified:

In[∘]:=
```
? TChangeDefaultCoords
```

Out[∘]=
```
Symbol
TChangeDefaultCoords[tensorID, coordinatesID] changes the default coordinate system of the tensor object tensorID to coordinatesID.
```

For example, let’s change the default coordinates of the perfect fluid stress tensor to spherical coordinates:

In[∘]:=
```
TChangeDefaultCoords["PerfectFluid", "Spherical"];
```

Now, when we display the tensor using `TList` without any arguments (or with just indices), this is the coordinate system that will be used:

In[∘]:=
```
TList["PerfectFluid"]
```

Out[∘]=

PerfectFluid:

$$\begin{aligned} T_{tt} &= \rho \\ T_{rr} &= p \\ T_{\theta\theta} &= p\, r^2 \\ T_{\phi\phi} &= p\, r^2\, \mathrm{Sin}[\theta]^2 \end{aligned}$$

Setting simplification assumptions

Often, coordinate transformations are only invertible for a specific range of coordinates. For example, let us define a new scalar in Minkowski space, which is equal to the spatial distance from the origin:

In[∘]:=
```
TShow@TNewTensor["SpatialDistance", "Minkowski", {}, {Sqrt[x^2 + y^2 + z^2]}, "d"]
```

Out[∘]=

SpatialDistance: $d\,(t, x, y, z) = \sqrt{x^2 + y^2 + z^2}$

When we convert this scalar to spherical coordinates, we get the expression $\sqrt{r^2}$, since Mathematica doesn’t automatically know that r is assumed to be non-negative:

In[∘]:=
```
TShow["SpatialDistance", "Spherical"]
```
Out[∘]=

$$\text{SpatialDistance:}\quad \mathrm{d}\,(t, r, \theta, \phi) = \sqrt{r^2}$$

As usual in Mathematica, such issues can be easily fixed by using `FullSimplify` with the correct **assumptions**. Most OGRe modules run their output through `FullSimplify` automatically, and the user may specify which assumptions to pass to `FullSimplify` using the module `TSimplifyAssumptions`:

In[∘]:=
```
? TSimplifyAssumptions
```
Out[∘]=

Symbol

TSimplifyAssumptions[*assumptions*] sets the assumptions to be used when simplifying expressions.

If *assumptions* is omitted, displays the currently used assumptions instead.

Use TSimplifyAssumptions[None] to clear previously set assumptions.

Note that these assumptions will be globally applied to **all** tensor calculations, which is usually the desired behavior, since for example the assumption $r \geq 0$ should apply to all tensors that use spherical coordinates. Let us set this assumption now:

In[∘]:=
```
TSimplifyAssumptions[r ≥ 0]
```

In fact, it is good practice to set any assumptions regarding the coordinates **as soon as they are defined**, so we should have set this assumption already when we defined the spherical coordinates in the beginning of this documentation. From now on, this assumption will automatically be used by modules that perform any kind of calculations on tensors. However, if we now try to show the scalar again using `TShow`, we still get the same (non-simplified) result:

In[∘]:=
```
TShow["SpatialDistance", "Spherical"]
```
Out[∘]=

$$\text{SpatialDistance:}\quad \mathrm{d}\,(t, r, \theta, \phi) = \sqrt{r^2}$$

The reason is that when OGRe calculates the components of a tensor in a particular representation, it calculates them **once and for all**, and then saves them in the object's data to be reused later. This is done to improve performance, so that the components don't have to be recalculated every time they are needed. In this case, since we already calculated the spatial distance in spherical coordinates when we showed it above - **before** we set the new simplification assumptions - that value has been saved, and will not be recalculated, even though we now have new assumptions.

However, we can force the simplification of the stored components with the new assumptions using the module `TSimplify`:

In[∘]:=
```
? TSimplify
```

Out[∘]=
```
Symbol

TSimplify[ID] simplifies all previously−calculated representations of the tensor object ID based on the user−defi⋱
    ned simplification assumptions set using TSimplifyAssumptions[ ]. To be used if the assumptions have chan⋱
    ged after the components have already been calculated.
```

In[∘]:=
```
TShow[TSimplify["SpatialDistance"], "Spherical"]
```

Out[∘]=

$$\text{SpatialDistance:}\quad \text{d}\,(t, r, \theta, \phi) = r$$

Here, again, note that `TSimplify` returns the ID of the tensor it simplifies, so we can compose it with `TShow` if we want to show that same tensor. However, since we are now using `TShow` with a second argument, we put `TSimplify[“SpatialDistance”]` as the first argument, instead of composing it directly with `@`.

Importing and exporting tensors

In a single Mathematica session, one can spend considerable time and effort defining tensors and doing various operations on them. However, as the tensors are only stored in memory, once the session is over and the kernel is stopped, all that information will be lost. Due to the non-linear nature of Mathematica notebooks, even if you saved the entire notebook, it can be hard or even impossible to retrace your steps and get the exact same tensors again from the information in the notebook.

Instead of defining all the tensors from scratch, OGRe allows the user to export tensors and then import them in another session to continue working with them later. The tensors are stored internally as an `Association`, and exporting a tensor essentially amounts to outputting the corresponding `Association`. **Warning: Do not change the exported data manually, as that might break the class invariants and cause errors after importing it back!**

To export a single tensor, use the `TExport` module:

In[∘]:=
```
? TExport
```

Out[∘]=
```
Symbol

TExport[ID] exports the raw tensor data for the tensor object ID as an Association.
```

For example, here is how the 4-velocity is stored internally:

In[∘]:=
```
TExport["4-Velocity"]
```

Out[∘]=
```
⟨|4-Velocity → ⟨|Components → ⟨|{{1}, Cartesian} → {1/√(1 - v²), v/√(1 - v²), 0, 0},
    {{-1}, Cartesian} → {-1/√(1 - v²), v/√(1 - v²), 0, 0}|⟩, DefaultCoords → Cartesian,
  DefaultIndices → {1}, Metric → Minkowski, Role → General, Symbol → u|⟩|⟩
```

This is a nested **`Association`**. The upper level has just one key: **`"4-Velocity"`**, which is the ID of the tensor. Its value is another **`Association`**, which has the following keys:

- **`"Components"`**: An **`Association`** containing the components of the tensor in different representations, each with a specific index configuration and coordinate system. The components are only generated when a particular combination of indices and coordinates is requested for the first time, so for example, here the components with both an upper and a lower index in Minkowski coordinates have been stored, but no components in spherical coordinates, since we have not tried to access them so far.
- **`"DefaultCoords"`**: The default coordinate system to use when displaying the tensor.
- **`"DefaultIndices"`**: The default index configuration to use when displaying the tensor.
- **`"Metric"`**: The unique ID of the metric that will be used to raise and lower the tensor's indices. Note that this is only a reference, so a tensor object with this ID must exist. If a tensor is exported, its metric must be exported separately as well for raising and lowering of indices to work.
- **`"Role"`**: The role of the tensor. Will be **`"Coordinates"`** if the tensor was created using **`TNewCoordinates`**, **`"Metric"`** if the tensor was created using **`TNewMetric`**, or **`"General"`** if the tensor was created using **`TNewTensor`**. Other roles are used internally by OGRe, such as **`"Temporary"`** for a temporary tensor created as an intermediate step in a calculation.
- **`"Symbol"`**: The symbol used to represent the tensor when displaying it.

Other keys also exist in special cases, such as **`"CoordTransformations"`** to store coordinate transformations defined using **`TAddCoordTransformation`**, as can be seen by exporting **`"Cartesian"`**:

In[∘]:=
```
TExport["Cartesian"]
```

Out[∘]=
```
⟨|Cartesian →
  ⟨|Components → ⟨|{{1}, Cartesian} → {t, x, y, z}|⟩, DefaultCoords → Cartesian,
   DefaultIndices → {1}, Role → Coordinates, Symbol → x, CoordTransformations →
     ⟨|Spherical → {x → r Cos[ϕ] Sin[θ], y → r Sin[θ] Sin[ϕ], z → r Cos[θ]}|⟩|⟩|⟩
```

To import a tensor back after it has been exported, use the **`TImport`** module:

In[∘]:=
```
? TImport
```

Out[∘]=
```
Symbol

TImport[data] imports a tensor that has been exported using TExport[ ].

WARNING:

1. The data is assumed to not have been manually modified by the user, so it is NOT checked for errors or inc⋱
      onsistencies. Importing tensor data that has been manually modified may cause errors or unexpected res⋱
      ults, and should be avoided.

2. The ID of the tensor will be taken from the name of the (single) key of the Association being imported. I⋱
      f a tensor with the same ID already exists, it will be overwritten.
```

To export **all** of the tensors defined in the current session, we may use the `TExportAll` module:

In[∘]:=
```
? TExportAll
```

Out[∘]=
```
Symbol

TExportAll[filename] exports the raw tensor data for all tensors defined in the current session as an Association.

filename is optional. If specified, the data is exported to a file with this name. If a full path is not given, the file w⋱
      ill be created in the current working directory, as given by Directory[ ]. Note that the file will be overwritten i⋱
      f it already exists.
```

The output will be an `Association` as above, where the keys are the names of all the tensors defined so far in the current session, and the value of each key is the data of that tensor. We will not show the complete output here, since it is very long, but let us just demonstrate that the keys of the `Association` are all of the tensors we defined so far in this session:

In[∘]:=
```
Keys[TExportAll[]]
```

Out[∘]=
```
{Cartesian, Spherical, Minkowski, Schwarzschild,
 Kretschmann, PerfectFluid, 4-Velocity, SpatialDistance}
```

The output can be saved in a Mathematica notebook, and imported using the module `TImportAll`:

```
In[∘]:= ? TImportAll
```

```
Out[∘]=
Symbol

TImportAll[source] imports tensor data that has been exported using TExportAll[ ].
If source is an Association, imports the data directly.
If source is a file name, imports the data from that file. If a full path is not given, the file should be located in t⋱
    he current working directory, as given by Directory[ ].
WARNING:
1. The data is assumed to not have been manually modified by the user, so it is NOT checked for errors or inc⋱
    onsistencies. Importing tensor data that has been manually modified may cause errors or unexpected res⋱
    ults, and should be avoided.
2. This will irreversibly delete ALL of the tensors already defined in the current session.
```

Note that `TImportAll` will delete any tensors already defined in the current session, whether or not they have the same ID as an imported tensor. To keep them, first export them into an `Association`, `Join` it with the `Association` you wish the import, and then use `TImportAll` on the result - or, alternatively, import the tensors one by one using `TImport`.

If a file name is given to `TExportAll`, the output will be saved to that file. If only the name of the file is given, and not a full path - e.g. `TExportAll["OGReTensors.m"]` - then the file will be saved in the current working directory, as given by `Directory[]`. To change the working directory, use `SetDirectory[]` before exporting the file. To import from the file, pass the file name to `TImportAll`, e.g. `TImportAll["OGReTensors.m"]`.

Finally, note that if for some reason you would like to delete all the tensors defined so far in the current session, you can simply import an empty `Association` as follows: `TImportAll[<||>]`. Be careful, as this action is **irreversible**!

Calculations with tensors

The TCalc module

Now that we have all the bookkeeping of tensors out of the way, we can finally discuss how to use those tensors in calculations. In OGRe, all tensor calculations are performed using the `TCalc` module:

In[∘]:=

```
? TCalc
```

Out[∘]=

Symbol

TCalc[*LHSTensorID*[*LHSIndices*], *RHSExpression*, *symbol*] calculates a tensor formula.

RHSExpression may include any number of tensors in the format *ID*[*indices*], where *ID* is a tensor object and *indices* is a string representing the order of indices, along with any combination of the following operations:

• Addition: For example, "A"["$\mu\nu$"] + "B"["$\mu\nu$"].

• Contraction: For example, "A"["$\mu\lambda$"] . "B"["$\lambda\nu$"].

• Multiplication by scalar: For example, 2 * "A"["$\mu\nu$"].

LHSTensorID specifies the ID of the tensor object in which to store the result. If omitted, the ID "Result" will be used.

LHSIndices specifies the order of indices of the resulting tensor. The indices must be a permutation of the free indices of *RHSExpression*. If omitted, the indices will be in the same order as they appear in *RHSExpression*. If *LHSTensorID* is omitted, then *LHSIndices* must be omitted as well.

symbol specifies the symbol to use for the result. If omitted, the placeholder symbol ⬚ will be used.

Any use of `TCalc` should be thought of as invoking a tensor equation of the form

$$L^{\alpha_1\cdots\alpha_p}_{\beta_1\cdots\beta_q} = R^{\alpha_1\cdots\alpha_p}_{\beta_1\cdots\beta_q},$$

where both the left-hand side and the right-hand side are tensors of the same rank and with the same **free indices** (that is, indices that are not being contracted). $L^{\alpha_1\cdots\alpha_p}_{\beta_1\cdots\beta_q}$ is the tensor that will be used to store the result, while $R^{\alpha_1\cdots\alpha_p}_{\beta_1\cdots\beta_q}$ is (the final result of) a general tensor calculation which contains any combination of addition, multiplication by scalar, trace, contraction, partial derivative, and covariant derivative. Let us now go over these operations one by one, and give some examples.

Addition of tensors

Addition of tensors in OGRe is represented by a sum of the form `“ID1”[“indices1”] + “ID2”[“indices2”]`, where `“ID1”` and `“ID2”` are the IDs of the tensor objects to be added, and `“indices1”` and `“indices2”` are the **index specifications** for each tensor, given as a string of letters. Note that you do **not** specify the position (upper or lower) of the indices. Furthermore, just like in any tensor equation, **the index letters themselves have no meaning**; they are just placeholders. Therefore, `“αβγ”`, `“abc”`, and `“ABC”` are all completely equivalent. The only requirement is that the **indices are consistent**; in the case of addition, this means that both tensors being added must have **the same indices up to permutation**.

The following constraints apply to addition of tensors:

- You may not add a tensor representing a coordinate system to any other tensor, since coordinates do not transform like tensors.
- You may not add two tensors associated with different metrics, since their sum would have undefined transformation properties.
- You may not add two tensors with different ranks, since that is not a well-defined operation.
- As stated above, both tensors must have the same indices up to permutation. $A^{\mu\nu} + B^{\mu\nu}$ and $A^{\mu\nu} + B^{\nu\mu}$ are both okay, but $A^{\mu\nu} + B^{\alpha\beta}$ doesn't make sense, as it has more free indices than the rank of the result.

As an example, let us add the Minkowski metric $\eta_{\mu\nu}$ and the perfect fluid stress tensor $T_{\mu\nu}$:

In[∘]:=
```
TShow@TCalc["Minkowski"["μν"] + "PerfectFluid"["μν"]]
```

Out[∘]=

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

Notice that the result was stored in a tensor with ID `"Result"`, and has no symbol. We can add a symbol to use as an additional argument:

In[∘]:=
```
TShow@TCalc["Minkowski"["μν"] + "PerfectFluid"["μν"], "S"]
```

Out[∘]=

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

With this symbol, the tensor equation we are calculating becomes:

$$S_{\mu\nu} = \eta_{\mu\nu} + T_{\mu\nu}.$$

We can also use a different ID for the result by giving it as the first argument, with or without a symbol:

In[∘]:=
```
TShow@TCalc["SumResult", "Minkowski"["μν"] + "PerfectFluid"["μν"], "S"]
TDelete["SumResult"]
```

Out[∘]=

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

(We deleted the result to avoid cluttering the tensor data with unused objects.) In the following examples, we will not specify a symbol, to keep the code cleaner.

Sometimes it is also helpful to specify indices for the result. To give an example, let us define the following non-symmetric tensor:

```
In[]:= TShow@TNewTensor["NonSymmetric", "Minkowski", {-1, -1},
         {{0, 0, 0, 1}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}}, "N"]
```

Out[]= NonSymmetric: $\mathrm{N}_{\mu\nu}(t,x,y,z) = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$

If we add it to the Minkowski metric, we get:

$$⬚_{\mu\nu} = \eta_{\mu\nu} + \mathrm{N}_{\mu\nu},$$

```
In[]:= TShow@TCalc["Minkowski"["μν"] + "NonSymmetric"["μν"]]
```

Out[]= Result: $⬚_{\mu\nu}(t,x,y,z) = \begin{pmatrix} -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

However, if we flip its index string from **"μν"** to **"νμ"**, then we instead get:

$$⬚_{\mu\nu} = \eta_{\mu\nu} + \mathrm{N}_{\nu\mu},$$

```
In[]:= TShow@TCalc["Minkowski"["μν"] + "NonSymmetric"["νμ"]]
```

Out[]= Result: $⬚_{\mu\nu}(t,x,y,z) = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$

Since the order of indices now matters, we can also define an index string for the left-hand side, to indicate the order of indices we want in the result. If that string is also **"νμ"**, then we get back to the original result:

$$⬚_{\nu\mu} = \eta_{\mu\nu} + \mathrm{N}_{\nu\mu},$$

In[∘]:=
```
TShow@TCalc["Result"["νμ"], "Minkowski"["μν"] + "NonSymmetric"["νμ"]]
```

Out[∘]=

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

We see that explicitly specifying the indices in `TCalc` allows it to have **a 1-to-1 correspondence with any tensor equation**. Importantly, note that there is no difference between `“NonSymmetric”[“μν”]` and `“NonSymmetric”[“νμ”]` on its own, as **the index labels themselves are meaningless** unless there is some context in which they obtain meaning - as is always the case for tensor expressions. However, there is a big difference between, for example, `“Minkowski”[“μν”] + “NonSymmetric”[“μν”]` and `“Minkowski”[“μν”] + “NonSymmetric”[“νμ”]`, as the indices have a different order, and thus the two expressions refer to adding different components.

Of course, any number of tensors can be added, not just two - and the same tensor can be added with different index configurations. For example, we can calculate:

$$\square_{\mu\nu} = \eta_{\mu\nu} + \mathrm{T}_{\mu\nu} + \mathrm{N}_{\mu\nu} + \mathrm{N}_{\nu\mu},$$

In[∘]:=
```
TShow@TCalc["Minkowski"["μν"] +
   "PerfectFluid"["μν"] + "NonSymmetric"["μν"] + "NonSymmetric"["νμ"]]
```

Out[∘]=

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

Multiplication of tensor by scalar

Multiplication of tensor by scalar in OGRe is represented by a product of the form `scalar * “ID”[“indices”]`, where `“ID”` is the ID of the tensor object to be multiplied, `“indices”` is an index specification as for addition, and `scalar` is the scalar to multiply by. Note that `scalar` should be a normal Mathematica symbol, such as a number or a variable, and **not** a tensor object of rank 0. To multiply a tensor by a tensor of rank 0, use contraction instead, as detailed in the next section.

As an example, let us multiply the Minkowski metric $\eta_{\mu\nu}$ by 2. The tensor equation we will be calculating is:

$$\square_{\mu\nu} = 2\,\eta_{\mu\nu},$$

and the OGRe expression to calculate it (and show the result) is:

In[ ]:=
```
TShow@TCalc[2 "Minkowski"["μν"]]
```

Out[ ]=

$$\text{Result: } ⬚_{\mu\nu}(t,x,y,z) = \begin{pmatrix} -2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

While in this example the indices seem redundant, they are necessary because in most non-trivial situations we would like to combine multiplication with other operations, such as addition or contraction, in which the order of indices matters. For example, consider:

$$⬚_{\mu\nu} = 2\,t\eta_{\mu\nu} - 3\,xT_{\mu\nu} + 4\,yN_{\mu\nu} - 5\,zN_{\nu\mu},$$

In[ ]:=
```
TShow@TCalc[2 t "Minkowski"["μν"] - 3 x "PerfectFluid"["μν"] +
   4 y "NonSymmetric"["μν"] - 5 z "NonSymmetric"["νμ"]]
```

Out[ ]=

$$\text{Result: } ⬚_{\mu\nu}(t,x,y,z) = \begin{pmatrix} -2t-3x\rho & 0 & 0 & 4y \\ 0 & 2t-3px & 0 & 0 \\ 0 & 0 & 2t-3px & 0 \\ -5z & 0 & 0 & 2t-3px \end{pmatrix}$$

Taking traces and contracting tensors: theoretical review

The most complicated tensor operation is **contraction**, a generalization of the vector inner product. This is done by summing over one or more disjoint pairs of indices, with each pair containing exactly one upper index and one lower index. Raising and lowering indices is one example of contraction: the metric (or its inverse) is contracted with a tensor. Coordinate transformations are another example, where we contract the Jacobian (or its inverse) with a tensor.

The simplest example of contraction is the **vector inner product**, which is defined as the contraction of a vector (one upper index) with a covector (one lower index):

$$v^{\mu} w_{\mu} = g_{\mu\nu} v^{\mu} w^{\nu} = g\,(v,\, w)\,.$$

The middle part of this equality comes from the fact that, as explained above, when we lower an index on w^{ν}, we use the metric:

$$w_{\mu} = g_{\mu\nu} w^{\nu}\,.$$

This, in turn, justifies the notation $g(v, w)$ on the right-hand side, as this is, in fact, an inner product of two vectors using the metric g (in index-free notation).

Contraction of indices in higher-rank tensors is simply a generalization of the inner product, for example:

$$A^{\mu\alpha} B_{\alpha\nu} = g_{\alpha\beta} A^{\mu\alpha} B^{\beta}{}_{\nu}.$$

We can also contract more than one index:

$$A^{\mu\nu} B_{\mu\nu} = g_{\mu\alpha} g_{\nu\beta} A^{\mu\nu} B^{\alpha\beta}.$$

This simply amount to the fact that lowering both indices of $B^{\alpha\beta}$ involves contracting each index with the metric. We can even contract two indices **of the same tensor**:

$$A^{\mu}{}_{\mu} = g_{\mu\nu} A^{\mu\nu}.$$

This is also called **taking the trace**. Furthermore, it is also possible to contract pairs of indices from more than two tensors at the same time:

$$A^{\mu\nu} B_{\nu\rho} C^{\rho\sigma} = g_{\nu\alpha} g_{\rho\beta} A^{\mu\nu} B^{\alpha\beta} C^{\rho\sigma}.$$

However, such operations can always be broken down into individual contractions of pairs of tensors. For example, in this case, one could first contract $B_{\nu\rho}$ with $C^{\rho\sigma}$ and then contract the result with $A^{\mu\nu}$- which is indeed how this kind of contraction will be performed in OGRe in practice:

$$A^{\mu\nu} B_{\nu\rho} C^{\rho\sigma} = A^{\mu\nu} \left(B_{\nu\rho} C^{\rho\sigma}\right).$$

In a contraction, there are two types of indices: **contracted indices**, which are summed upon, and **free indices**, which are not summed upon. The rank of the tensor that results from the contraction is the number of free indices. So for example, in the expression $A^{\mu\alpha} B_{\alpha\nu}$ we have one contracted index, α, and two free indices, μ and ν. Therefore, the resulting tensor is of rank two: $T^{\mu}{}_{\nu} = A^{\mu\alpha} B_{\alpha\nu}$.

Taking traces and contracting tensors: OGRe syntax

Contraction of tensors in OGRe is represented by an expression of the form `"ID1"["indices1"] . "ID2"["indices2"]`, where `"ID1"` and `"ID2"` are the IDs of the tensor objects to be contracted, and `"indices1"` and `"indices2"` are the index strings for each tensor. Any **matching indices** in both index strings will be contracted. This means that, for example, $v^{\mu} w_{\mu}$ is calculated using `"v"["μ"] . "w"["μ"]` and $A^{\mu\nu} B_{\nu\rho} C^{\rho\sigma}$ is calculated using `"A"["μν"] . "B"["νρ"] . "C"["ρσ"]`. Note that the user doesn't need to worry about the contracted indices being one upper and one lower, which is a common source of errors when contracting tensors by hand; the order of the indices, and whether the same index repeats twice, is all that matters.

As a first example, let us create the stress-energy tensor for a perfect fluid with a 4-velocity u^{μ}. This is defined as follows:

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

Even though this does not involve any contractions, it still counts as a "trivial" contraction, since two tensors (the 4-velocities) are juxtaposed next to each other to create another tensor. This is also known as an **outer product**. Therefore, it uses the same dot product syntax as any other contraction, except that there are **no matching indices**. Note that this expression involves not just contraction (in the first term), but also multiplication by scalar (in both terms), and addition of the two terms together. Again, OGRe takes care of everything behind the scene, so this just works:

In[∘]:=
```
TShow@TCalc["PerfectFluid",
  (ρ + p) "4-Velocity"["μ"]."4-Velocity"["ν"] + p "Minkowski"["μν"], "T"]
```

Out[∘]=

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

Indeed, for $v = 0$ we get the previously defined stress tensor:

In[∘]:=
```
TShow["PerfectFluid"] /. v → 0
```

Out[∘]=

$$\text{PerfectFluid:}\quad T_{\mu\nu}(t, x, y, z) = \begin{pmatrix} \rho & 0 & 0 & 0 \\ 0 & p & 0 & 0 \\ 0 & 0 & p & 0 \\ 0 & 0 & 0 & p \end{pmatrix}$$

Multiplying a tensor by a scalar tensor (i.e. a tensor of rank 0) is also done using a “trivial” contraction with no contracted indices. For example:

In[∘]:=
```
TShow[TCalc["SpatialDistance"[""]."Minkowski"["μν"]], "Spherical"]
```

Out[∘]=

$$\text{Result:}\quad ⬚_{\mu\nu}(t, r, \theta, \phi) = \begin{pmatrix} -r & 0 & 0 & 0 \\ 0 & r & 0 & 0 \\ 0 & 0 & r^3 & 0 \\ 0 & 0 & 0 & r^3\,\text{Sin}[\theta]^2 \end{pmatrix}$$

Note the empty index string `[“”]`, which is mandatory in order for OGRe to recognize that the scalar is a tensor object. We can also multiply a scalar by another scalar:

In[∘]:=
```
TShow[TCalc["SpatialDistance"[""]."SpatialDistance"[""]], "Spherical"]
```

Out[∘]=

$$\text{Result:}\quad ⬚\,(t, r, \theta, \phi) = r^2$$

Now let us demonstrate some non-trivial contractions. First, we have the inner product of vectors - in this case, we get the norm (squared) of the 4-velocity, since we are contracting it with itself:

In[∘]:=
```
TShow@TCalc["4-Velocity"["μ"]."4-Velocity"["μ"]]
```

Out[∘]=

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

We can also contract several tensors together, with **two** matching pairs of indices:

In[∘]:=
```
TShow@TCalc["4-Velocity"["μ"]."PerfectFluid"["μν"]."NonSymmetric"["νρ"]]
```

Out[∘]=

$$\text{Result:}\quad \square_\mu(t, x, y, z) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ -\frac{\rho}{\sqrt{1-v^2}} \end{pmatrix}$$

Finally, to take the trace of a tensor, we simply match pairs of indices in that tensor's index string:

In[∘]:=
```
TShow@TCalc["Minkowski"["μμ"]]
```

Out[∘]=

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

In[∘]:=
```
TShow@TCalc["PerfectFluid"["μμ"]]
```

Out[∘]=

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

Of course, this also works for tensors with more than two indices, as we will see below. Any combination of indices can be used, with no limit on the number of traces taken for each tensor.

Derivatives and curvature tensors

The **partial derivative** ∂_μ is represented in OGRe using the symbol `TPartialD`. It can be contracted with other tensors using the usual OGRe contraction notation - including an appropriate index string - to calculate gradients and divergences.

In[∘]:=
```
? TPartialD
```

Out[∘]=

Symbol

TPartialD[*index*] represents the partial derivative when used in a tensor expression given to TCalc[ ].

The **gradient** of a tensor is the partial derivative ∂_μ acting on the tensor with a free index, e.g. $\partial_\mu \phi$, $\partial_\mu v^\nu$, or $\partial_\mu T^{\nu\lambda}$, resulting in a tensor of one higher rank. The **divergence** of a tensor is the contraction of the partial derivative ∂_μ with one of the tensor's indices, e.g. $\partial_\mu v^\mu$ or $\partial_\mu T^{\mu\nu}$, resulting in a tensor of one lower rank. The syntax for both is the same; if the index specification of `TPartialD["μ"]` matches one of the indices of the tensor to its right, then the divergence will be calculated, otherwise the gradient will be calculated. Divergences using the partial derivative are not commonly used in general relativity, so we will not deal with them here.

Let us now consider some commonly-used curvature tensors and how to calculate them using the partial derivative in OGRe.

The Christoffel symbols

The **Christoffel symbols** are a very important quantity in differential geometry. They are the coefficients of the **Levi-Civita connection,** which is the unique torsion-free connection that preserves the metric. The Christoffel symbols are defined as follows:

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

Each of the terms inside the parentheses is a gradient of the metric, with different indices. For example, the first term $\partial_\mu \mathbf{g}_{\nu\sigma}$ is represented in OGRe as `TPartialD["μ"].“Metric”["νσ"]` where `“Metric”` is the tensor object representing the metric. Since contraction, addition, and multiplication by scalar can be combined arbitrarily when using `TCalc`, we can calculate the Christoffel symbols in a straightforward way as follows:

In[∘]:=
```
TList@
 TChangeDefaultIndices[TCalc["SchwarzschildChristoffel", 1/2 "Schwarzschild"["λσ"].
     (TPartialD["μ"]."Schwarzschild"["νσ"] + TPartialD["ν"]."Schwarzschild"["σμ"] -
       TPartialD["σ"]."Schwarzschild"["μν"]), "Γ"], {1, -1, -1}]
```

Out[∘]=

SchwarzschildChristoffel:

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

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

$$\Gamma^{r}{}_{rr} = \frac{M}{2Mr-r^2}$$

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

$$\Gamma^{r}{}_{\phi\phi} = (2M-r)\,\mathrm{Sin}[\theta]^2$$

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

$$\Gamma^{\theta}{}_{\phi\phi} = -\mathrm{Cos}[\theta]\,\mathrm{Sin}[\theta]$$

$$\Gamma^{\phi}{}_{\theta\phi} = \Gamma^{\phi}{}_{\phi\theta} = \mathrm{Cot}[\theta]$$

Importantly, **the Christoffel symbols are not a tensor**, meaning that they do not transform as a tensor does, so raising and lowering their indices, and especially transforming them to a different coordinate system, may have unexpected results. Therefore, just as when doing calculations by hand, it is recommended to handle the Christoffel symbols with caution, ensuring that you only ever use their representation with the first index up and the other two indices down (which is why I used `TChangeDefaultIndices` here), and never transform them to a different coordinate system.

OGRe also provides a built-in module called `TChristoffel` to automatically calculate the Christoffel symbols for a given metric. Essentially, this module does exactly what we did above: calculates the formula using `TCalc`, names the resulting tensor "`[metricID]Christoffel`", gives it the symbol Γ, and set its default indices to `{1,-1,-1}`.

In[∘]:=
```
? TChristoffel
```

Out[∘]=

Symbol

TChristoffel[*metricID*] calculates the Christoffel symbols from the metric *metricID* and stores the result in a ne
w tensor object with ID "*metricID*Christoffel".

As an example of using this module, let us define the **Friedmann–Lemaitre–Robertson–Walker**

(FLRW) metric, which describes an expanding universe:

In[∘]:=
```
TInitializeSymbols[a, k];
TShow@TNewMetric["FLRW", "Spherical",
  DiagonalMatrix[{-1, a[t]^2/(1 - k r^2), a[t]^2 r^2, a[t]^2 r^2 Sin[θ]^2}]]
```

Out[∘]=

$$\text{FLRW:}\quad g_{\mu\nu}(t,r,\theta,\phi) = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & \frac{a[t]^2}{1-k\,r^2} & 0 & 0 \\ 0 & 0 & r^2\,a[t]^2 & 0 \\ 0 & 0 & 0 & r^2\,a[t]^2\,\text{Sin}[\theta]^2 \end{pmatrix}$$

Here, $a\,(t)$ is the **scale factor** and k is the curvature of the spatial surfaces, with $k = +1, 0, -1$ corresponding to positively curved, flat, or negatively curved respectively. Its Christoffel symbols can be easily calculated using `TChristoffel`:

In[∘]:=
```
TList@TChristoffel["FLRW"]
```

Out[∘]=

FLRWChristoffel:

$$\Gamma^{t}{}_{rr} = \frac{a[t]\,a'[t]}{1-k\,r^2}$$

$$\Gamma^{t}{}_{\theta\theta} = r^2\,a[t]\,a'[t]$$

$$\Gamma^{t}{}_{\phi\phi} = r^2\,a[t]\,\text{Sin}[\theta]^2\,a'[t]$$

$$\Gamma^{r}{}_{tr} = \Gamma^{r}{}_{rt} = \Gamma^{\theta}{}_{t\theta} = \Gamma^{\theta}{}_{\theta t} = \Gamma^{\phi}{}_{t\phi} = \Gamma^{\phi}{}_{\phi t} = \frac{a'[t]}{a[t]}$$

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

$$\Gamma^{r}{}_{\theta\theta} = r\left(-1+k\,r^2\right)$$

$$\Gamma^{r}{}_{\phi\phi} = r\left(-1+k\,r^2\right)\text{Sin}[\theta]^2$$

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

$$\Gamma^{\theta}{}_{\phi\phi} = -\text{Cos}[\theta]\,\text{Sin}[\theta]$$

$$\Gamma^{\phi}{}_{\theta\phi} = \Gamma^{\phi}{}_{\phi\theta} = \text{Cot}[\theta]$$

The Riemann tensor

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

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

To calculate it, we can simply write down the formula in `TCalc` with the correct index contracted. Note that this time we specified the LHS indices explicitly, since they are not in the same order as the RHS indices in our definition:

In[∘]:=

```
TList@TChangeDefaultIndices[TCalc["SchwarzschildRiemann"["ρσμν"],
   TPartialD["μ"]."SchwarzschildChristoffel"["ρνσ"] -
    TPartialD["ν"]."SchwarzschildChristoffel"["ρμσ"] +
    "SchwarzschildChristoffel"["ρμλ"]."SchwarzschildChristoffel"["λνσ"] -
    "SchwarzschildChristoffel"["ρνλ"].
     "SchwarzschildChristoffel"["λμσ"], "R"], {1, -1, -1, -1}]
```

Out[∘]=

SchwarzschildRiemann:

$$\begin{array}{lcl}
R^{t}{}_{rtr} = -R^{t}{}_{rrt} & = & \frac{2M}{r^2(-2M+r)} \\
R^{t}{}_{\theta t\theta} = R^{r}{}_{\theta r\theta} = -R^{t}{}_{\theta\theta t} = -R^{r}{}_{\theta\theta r} & = & -\frac{M}{r} \\
R^{t}{}_{\phi t\phi} = R^{r}{}_{\phi r\phi} = -R^{t}{}_{\phi\phi t} = -R^{r}{}_{\phi\phi r} & = & -\frac{M\,\mathrm{Sin}[\theta]^2}{r} \\
R^{r}{}_{ttr} & = & \frac{2M(-2M+r)}{r^4} \\
R^{r}{}_{trt} & = & \frac{2M(2M-r)}{r^4} \\
R^{\theta}{}_{tt\theta} = R^{\phi}{}_{tt\phi} & = & \frac{M(2M-r)}{r^4} \\
R^{\theta}{}_{t\theta t} = R^{\phi}{}_{t\phi t} & = & \frac{M(-2M+r)}{r^4} \\
R^{\theta}{}_{rr\theta} = R^{\phi}{}_{rr\phi} & = & \frac{M}{r^2(-2M+r)} \\
R^{\theta}{}_{r\theta r} = R^{\phi}{}_{r\phi r} & = & \frac{M}{(2M-r)\,r^2} \\
R^{\theta}{}_{\phi\theta\phi} = -R^{\theta}{}_{\phi\phi\theta} & = & \frac{2M\,\mathrm{Sin}[\theta]^2}{r} \\
R^{\phi}{}_{\theta\theta\phi} = -R^{\phi}{}_{\theta\phi\theta} & = & -\frac{2M}{r}
\end{array}$$

The Riemann tensor with all its indices down satisfies the following symmetry and antisymmetry relations:

$$R_{\rho\sigma\mu\nu} = -R_{\sigma\rho\mu\nu} = -R_{\rho\sigma\nu\mu} = R_{\mu\nu\rho\sigma}.$$

We can verify this for the Schwarzschild Riemann tensor using `TList`, as it automatically detects components that are the same up to sign:

In[∘]:=
```
TList["SchwarzschildRiemann", {-1, -1, -1, -1}]
```

Out[∘]=

SchwarzschildRiemann:

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

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

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

$$R_{t\phi t\phi} = R_{\phi t\phi t} = \frac{M(-2M+r)\,\mathrm{Sin}[\theta]^2}{r^2}$$

$$R_{t\phi\phi t} = R_{\phi tt\phi} = \frac{M(2M-r)\,\mathrm{Sin}[\theta]^2}{r^2}$$

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

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

$$R_{r\phi r\phi} = R_{\phi r\phi r} = -R_{r\phi\phi r} = \frac{M\,\mathrm{Sin}[\theta]^2}{2M-r}$$

$$R_{\theta\phi\theta\phi} = R_{\phi\theta\phi\theta} = -R_{\theta\phi\phi\theta} = -R_{\phi\theta\theta\phi} = 2Mr\,\mathrm{Sin}[\theta]^2$$

$$R_{\phi rr\phi} = \frac{M\,\mathrm{Sin}[\theta]^2}{-2M+r}$$

We can also use the shorthand module `TRiemannTensor`:

In[∘]:=
```
? TRiemannTensor
```

Out[∘]=

Symbol

TRiemannTensor[*metricID*] calculates the Riemann tensor from the metric *metricID* and stores the result in a ne⋱

w tensor object with ID "*metricID*Riemann".

If a tensor with ID "*metricID*Christoffel" exists, it will be assumed to be the Christoffel symbols of the metric, a⋱

nd will be used in the calculation. Otherwise, "*metricID*Christoffel" will be created using TChristoffel[ ].

For example, for the FLRW metric:

In[ ]:=
```
TList@TRiemannTensor["FLRW"]
```

Out[ ]=

FLRWRiemann:

$$R^{t}{}_{rtr} = \frac{a[t]\,a''[t]}{1-k\,r^2}$$

$$R^{t}{}_{rrt} = \frac{a[t]\,a''[t]}{-1+k\,r^2}$$

$$R^{t}{}_{\theta t\theta} = -R^{t}{}_{\theta\theta t} = r^2\,a[t]\,a''[t]$$

$$R^{t}{}_{\phi t\phi} = -R^{t}{}_{\phi\phi t} = r^2\,a[t]\,\mathrm{Sin}[\theta]^2\,a''[t]$$

$$R^{r}{}_{ttr} = R^{\theta}{}_{tt\theta} = R^{\phi}{}_{tt\phi} = -R^{r}{}_{trt} = -R^{\theta}{}_{t\theta t} = -R^{\phi}{}_{t\phi t} = \frac{a''[t]}{a[t]}$$

$$R^{r}{}_{\theta r\theta} = R^{\phi}{}_{\theta\phi\theta} = -R^{r}{}_{\theta\theta r} = -R^{\phi}{}_{\theta\theta\phi} = r^2\left(k + a'[t]^2\right)$$

$$R^{r}{}_{\phi r\phi} = R^{\theta}{}_{\phi\theta\phi} = -R^{r}{}_{\phi\phi r} = -R^{\theta}{}_{\phi\phi\theta} = r^2\,\mathrm{Sin}[\theta]^2\left(k + a'[t]^2\right)$$

$$R^{\theta}{}_{rr\theta} = R^{\phi}{}_{rr\phi} = \frac{k+a'[t]^2}{-1+k\,r^2}$$

$$R^{\theta}{}_{r\theta r} = R^{\phi}{}_{r\phi r} = \frac{k+a'[t]^2}{1-k\,r^2}$$

Finally, recall that above, we gave the **Kretschmann scalar** for the Schwarzschild metric as an example of a scalar. Now that we have the Riemann tensor, and the ability to contract tensors, we can actually calculate the Kretschmann scalar from scratch. The formula is:

$$K = R_{\rho\sigma\mu\nu}\,R^{\rho\sigma\mu\nu},$$

so it can be easily calculated in OGRe as follows:

In[ ]:=
```
TShow@TCalc["Kretschmann",
  "SchwarzschildRiemann"["ρσμν"]."SchwarzschildRiemann"["ρσμν"], "K"]
```

Out[ ]=

$$\text{Kretschmann:}\quad K\,(t, r, \theta, \phi) = \frac{48\,M^2}{r^6}$$

The Ricci tensor and scalar

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

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

Therefore, we can calculate it by taking the trace, with the usual `TCalc` syntax. For the Schwarzschild metric, the Ricci tensor vanishes:

```
In[∘]:= TList@TCalc["SchwarzschildRiemann"["λμλν"], "R"]
```

```
Out[∘]= Result:
        No non–zero elements.
```

We can also use the shorthand module `TRicciTensor`:

```
In[∘]:= ? TRicciTensor
```

```
Out[∘]= Symbol

TRicciTensor[metricID] calculates the Ricci tensor from the metric metricID and stores the result in a new tensor o⋱
        bject with ID "metricIDRicciTensor".
If a tensor with ID "metricIDRiemann" exists, it will be assumed to be the Riemann tensor of the metric, an⋱
        d will be used in the calculation. Otherwise, "metricIDRiemann" will be created using TRiemannTensor[ ].
```

Here is the Ricci tensor for the FLRW metric:

```
In[∘]:= TList@TRicciTensor["FLRW"]
```

Out[∘]= FLRWRicciTensor:

$$R_{tt} = -\frac{3\,a''[t]}{a[t]}$$

$$R_{rr} = \frac{2\left(k+a'[t]^2\right)+a[t]\,a''[t]}{1-k\,r^2}$$

$$R_{\theta\theta} = r^2\left(2\left(k+a'[t]^2\right)+a[t]\,a''[t]\right)$$

$$R_{\phi\phi} = r^2\,\mathrm{Sin}[\theta]^2\left(2\left(k+a'[t]^2\right)+a[t]\,a''[t]\right)$$

The **Ricci scalar** is the trace of the Ricci tensor:

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

We can calculate it from the Ricci tensor by taking the trace:

```
In[∘]:= TShow@TCalc["FLRWRicciTensor"["μμ"], "R"]
```

Out[∘]=

$$\text{Result:}\quad R\,(t, r, \theta, \phi) = \frac{6\left(k+a'[t]^2+a[t]\,a''[t]\right)}{a[t]^2}$$

Or, as usual, we can simply use the shorthand module `TRicciScalar` to calculate it directly from the metric:

In[∘]:=
```
? TRicciScalar
```

Out[∘]=
```
Symbol
TRicciScalar[metricID] calculates the Ricci scalar from the metric metricID and stores the result in a new tensor o⋱
    bject with ID "metricIDRicciScalar".
If a tensor with ID "metricIDRicciTensor" exists, it will be assumed to be the Ricci tensor of the metric, and w⋱
    ill be used in the calculation. Otherwise, "metricIDRicciTensor" will be created using TRicciTensor[ ].
```

In[∘]:=
```
TList@TRicciScalar["FLRW"]
```

Out[∘]=

FLRWRicciScalar:

$$R = \frac{6\left(k+a'[t]^2+a[t]\,a''[t]\right)}{a[t]^2}$$

The Einstein tensor

The **Einstein tensor** $G_{\mu\nu}$ is given by:

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

As with all other curvature tensors, we can calculate it by combining the previously calculated tensors with the usual syntax:

In[∘]:=
```
TList@TCalc["FLRWRicciTensor"["μν"] - 1/2 "FLRW"["μν"].“FLRWRicciScalar”[""], "G"]
```

Out[∘]=

Result:

$$G_{tt} = \frac{3\left(k+a'[t]^2\right)}{a[t]^2}$$

$$G_{rr} = \frac{k+a'[t]^2+2\,a[t]\,a''[t]}{-1+k\,r^2}$$

$$G_{\theta\theta} = -r^2\left(k+a'[t]^2+2\,a[t]\,a''[t]\right)$$

$$G_{\phi\phi} = -r^2\,\mathrm{Sin}[\theta]^2\left(k+a'[t]^2+2\,a[t]\,a''[t]\right)$$

Or we can use the built-in module `TEinsteinTensor`:

```
In[∘]:= ? TEinsteinTensor
```

```
Out[∘]=
Symbol

TEinsteinTensor[metricID] calculates the Einstein tensor from the metric metricID and stores the result in a new t⋱
    ensor object with ID "metricIDEinstein".
If a tensor with ID "metricIDRicciTensor" exists, it will be assumed to be the Ricci tensor of the metric, and w⋱
    ill be used in the calculation. Otherwise, "metricIDRicciTensor" will be created using TRicciTensor[ ].
```

```
In[∘]:= TList@TEinsteinTensor["FLRW"]
```

Out[∘]=

FLRWEinstein:

$$
\begin{aligned}
G_{tt} &= \frac{3\left(k+a'[t]^2\right)}{a[t]^2}\\
G_{rr} &= \frac{k+a'[t]^2+2\,a[t]\,a''[t]}{-1+k\,r^2}\\
G_{\theta\theta} &= -r^2\left(k+a'[t]^2+2\,a[t]\,a''[t]\right)\\
G_{\phi\phi} &= -r^2\,\mathrm{Sin}[\theta]^2\left(k+a'[t]^2+2\,a[t]\,a''[t]\right)
\end{aligned}
$$

Covariant derivatives

The partial derivative has limited use in general relativity, as **it does not transform like a tensor**. Therefore, it is only used in special cases, such as calculating the Christoffel symbols and the Riemann tensor. The **covariant derivative** ∇_μ is a generalization of the partial derivative, which does transform like a tensor (as long as it acts on a proper tensor). It is defined as follows:

- On a scalar Φ, the covariant derivative acts as $\nabla_\mu \Phi \equiv \partial_\mu \Phi$.
- On a vector v^μ, the covariant derivative acts as $\nabla_\mu v^\nu \equiv \partial_\mu v^\nu + \Gamma^\nu_{\mu\lambda} v^\lambda$.
- On a covector w_μ, the covariant derivative acts as $\nabla_\mu w_\nu \equiv \partial_\mu w_\nu - \Gamma^\lambda_{\mu\nu} u_\lambda$.

More generally, on a rank (p, q) tensor with components $T^{\nu_1 \cdots \nu_p}_{\sigma_1 \cdots \sigma_q}$, the covariant derivative $\nabla_\mu T^{\nu_1 \cdots \nu_p}_{\sigma_1 \cdots \sigma_q}$ is defined as follows:

- The first term will be $\partial_\mu T^{\nu_1 \cdots \nu_p}_{\sigma_1 \cdots \sigma_q}$.
- We **add** one term $\Gamma^{\nu_i}_{\mu\lambda} T^{\nu_1 \cdots \lambda \cdots \nu_p}_{\sigma_1 \cdots \sigma_q}$ for each upper index ν_i.
- We **subtract** one term $\Gamma^\lambda_{\mu\sigma_i} T^{\nu_1 \cdots \nu_p}_{\sigma_1 \cdots \lambda \cdots \sigma_q}$ for each lower index σ_i.

Note that even though the covariant derivative is made from ingredients that do not transform like tensors - the partial derivative and the Christoffel symbols - the unwanted terms in the transformations of these ingredients cancel each other exactly, so that in the end, the entire sum does transform like a tensor.

As usual, we can, of course, write down the covariant derivative manually. For example, the covariant divergence of the metric is:

$$\nabla_\mu g_{\alpha\beta} = \partial_\mu g_{\alpha\beta} - \Gamma^\lambda_{\mu\alpha} g_{\lambda\beta} - \Gamma^\lambda_{\mu\beta} g_{\alpha\lambda}.$$

It should vanish, by definition, for any metric; this is what we meant when we said the Levi-Civita connection **preserves** the metric. Indeed, we have for the Schwarzschild metric:

In[∘]:=
```
TList@TCalc[TPartialD["μ"].“Schwarzschild"["αβ"] -
   "SchwarzschildChristoffel"["λμα"].“Schwarzschild"["λβ"] -
   "SchwarzschildChristoffel"["λμβ"].“Schwarzschild"["αλ"]]
```

Out[∘]=
```
Result:
No non–zero elements.
```

Much more conveniently, the covariant derivative is represented in OGRe as `TCovariantD`. It will automatically add the correct terms, as detailed above, for each of the tensor's indices. To use it, simply contract it with any tensor, just like `TPartialD`:

In[∘]:=
```
? TCovariantD
```

Out[∘]=

Symbol

TCovariantD[*index*] represents the covariant derivative when used in a tensor expression given to TCalc[ ].

If a tensor with ID "*metricID*Christoffel" exists, where *metricID* is the metric associated with the tensor the deriva⋱

tive is acting upon, then it will be assumed to be the Levi–Civita connection of the metric, and will be used i⋱

n the calculation. Otherwise, "*metricID*Christoffel" will be created using TChristoffel[ ].

For example, we can check that the covariant derivative of the FLRW metric also vanishes:

In[∘]:=
```
TList@TCalc[TCovariantD["μ"].“FLRW"["αβ"]]
```

Out[∘]=
```
Result:
No non–zero elements.
```

The covariant divergence of the Einstein tensor is:

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

Note that it involves a contraction in the index μ, which becomes a trace in the first Christoffel symbol. This expression vanishes because of the **Bianchi identity**:

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

To calculate it in OGRe, we simply write:

```
In[]:= TList@TCalc[TCovariantD["μ"]."FLRWEinstein"["μν"]]
```

```
Out[]= Result:
       No non–zero elements.
```

Finally, for a non-trivial result, let us recall that the stress-energy tensor should be **conserved**:

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

This follows from the fact that $\nabla_\mu G^{\mu\nu} = 0$, combined with the **Einstein equation**:

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

where $\kappa = 1$ or $\kappa = 8\pi$ depending on your preferred units. However, unlike $\nabla_\mu G^{\mu\nu} = 0$, the relation $\nabla_\mu T^{\mu\nu} = 0$ is **not** an identity; it is an **energy-momentum conservation equation**. To derive the equation for the FLRW metric, let us first define the rest-frame fluid 4-velocity in this spacetime:

```
In[]:= TShow@TNewTensor["RestVelocity", "FLRW", {1}, {1, 0, 0, 0}, "u"]
```

Out[]= RestVelocity: $u^\mu(t, r, \theta, \phi) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$

Using the 4-velocity and the metric, we redefine the perfect fluid stress tensor in the FLRW spacetime using the formula $T^{\mu\nu} = (\rho + p)\, u^\mu u^\nu + p g^{\mu\nu}$, and give ρ and p spacetime dependence:

In[•]:=
```
TCalc["PerfectFluid",
  (ρ[t, r, θ, ϕ] + p[t, r, θ, ϕ]) "RestVelocity"["μ"]."RestVelocity"["ν"] +
   p[t, r, θ, ϕ] "FLRW"["μν"], "T"];
TChangeDefaultIndices["PerfectFluid", {1, 1}];
TShow["PerfectFluid"]
```

Out[•]=

$$\text{PerfectFluid:}\quad T^{\mu\nu}(t, r, \theta, \phi) =$$

$$\begin{pmatrix} \rho[t, r, \theta, \phi] & 0 & 0 & 0 \\ 0 & \frac{(1-k\,r^2)\,p[t,r,\theta,\phi]}{a[t]^2} & 0 & 0 \\ 0 & 0 & \frac{p[t,r,\theta,\phi]}{r^2\,a[t]^2} & 0 \\ 0 & 0 & 0 & \frac{\text{Csc}[\theta]^2\,p[t,r,\theta,\phi]}{r^2\,a[t]^2} \end{pmatrix}$$

Finally, we take the covariant derivative of the stress tensor:

In[•]:=
```
TList@TCalc["FLRWConservation", TCovariantD["μ"]."PerfectFluid"["μν"]]
```

Out[•]=

$$\text{FLRWConservation:}$$

$$\square^{t} = \frac{3\,(p[t,r,\theta,\phi]+\rho[t,r,\theta,\phi])\,a'[t]}{a[t]} + \rho^{(1,0,0,0)}[t, r, \theta, \phi]$$

$$\square^{r} = \frac{(1-k\,r^2)\,p^{(0,1,0,0)}[t,r,\theta,\phi]}{a[t]^2}$$

$$\square^{\theta} = \frac{p^{(0,0,1,0)}[t,r,\theta,\phi]}{r^2\,a[t]^2}$$

$$\square^{\phi} = \frac{\text{Csc}[\theta]^2\,p^{(0,0,0,1)}[t,r,\theta,\phi]}{r^2\,a[t]^2}$$

From demanding that the *t* component vanishes, we get the following equation:

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

We see that in an expanding universe, energy is not conserved, but rather, the energy density changes with time in a way that depends on the scale factor. If the universe is not expanding, that is, $\dot{a} = 0$, then energy will be conserved.

Additional information

Compatibility

This package is compatible with Mathematica 12.0 or newer, on any platform (Windows, Linux, and Mac). It should also run on Mathematica 11 or 10, but that has not been tested. The package will **not** work on versions prior to Mathematica 10, due to extensive use of `Association`, which was introduced in Mathematica 10.0.

Version history

- Version 1.0 (2021-02-10)
  - Initial release.

Future plans

This package is a work in progress. The following features are currently planned for future versions:

- Get a subset of a tensor's components by replacing one or more indices with a specific coordinate, e.g. T^{012} will return that component and $T^{\mu\nu 0}$ will return a rank-2 tensor.
- Show relations between tensors, e.g. which tensors are using a particular metric/coordinates or which metric/coordinates are being used by a particular tensor.
- Export and import user settings, such as default index letters and simplification assumptions.
- More control over the output style of `TShow` and `TList`.
- More modules to calculate commonly-used curvature tensors, such as the Kretschmann scalar and the Weyl tensor.
- Export the raw components of a tensor in a specific representation.
- Parallelization of various operations for increased performance.
- Comma and semicolon operators as alternative to `TPartialD` and `TCovariantD`.
- Disallow coordinate transformations and raising/lowering indices on the Christoffel symbols, since they are not a tensor.
- `TShow["ID"["indices"]]` should use the given indices when showing the tensor, instead of the default indices.
- Expressions like `TCalc["A"+"B"]` should automatically deduce the indices to use.
- `TCalc["A"."B"]` should also automatically deduce the indices to use, e.g. if one of the tensors is a scalar or both are vectors.
- Use any Mathematica symbols, instead of just strings, as tensor IDs - for less cluttered syntax.
- Define a tensor in a specific coordinate system, instead of the default coordinates of the metric.

- Improve performance of raising and lowering indices by doing them all at once instead of index by index.
- Support for connections with torsion.
- If three coordinate systems are defined, and we know how to get from 1 to 2 and from 2 to 3, then OGRe should automatically deduce the transformation from 1 to 3.
- Integrate documentation with the Wolfram Documentation Center.
- And much more...

If you would like a request any additional features, or if you encounter any bugs, please feel free to open a new issue on GitHub!

Citing

If you use this package in your research, please cite it as follows:

- Barak Shoshany, OGRe: An Object-Oriented General Relativity Toolkit for Mathematica, https://github.com/bshoshany/OGRe (2021).

(This citation will be replaced with a journal reference once a paper is published.)

Copyright and license

Acknowledgements

A major portion of the code for this package was written while I was a postdoctoral researcher in Niayesh Afshordi’s group at Perimeter Institute for Theoretical Physics in Waterloo, Ontario, Canada. I would like to thank Niayesh Afshordi and Perimeter Institute for their support during that time. Research at Perimeter Institute is supported in part by the Government of Canada through the Department of Innovation, Science, and Economic Development Canada and by the Province of Ontario through the Ministry of Colleges and Universities.