OGRe: An **O**bject-Oriented **G**eneral **Re**lativity Package for Mathematica

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

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 below.

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. Metrics can also be displayed as a line element.

- Automatically simplify tensor components, optionally with user-defined simplification assumptions. Simplifications can be parallelized for a significant performance boost. The user may utilize the package's simplification algorithm for any Mathematica expression, not just tensors.

- 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.

- Highly customizable. User settings are exported and imported along with the tensors. Some settings are persistent between sessions.

- 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, Einstein tensor, curve Lagrangian, and volume element from a metric.

- Calculate the norm-squared of tensors of any rank.

- Calculate the geodesic equations in terms of a user-defined affine curve parameter, in two different ways: from the Christoffel symbols or from the curve Lagrangian. For spacetime metrics, the geodesic equations can be calculated in terms of the time coordinate.

- Designed with speed and performance in mind, using optimized algorithms designed specifically for this package.

- Fully portable. Can be imported directly from the web into any Mathematica notebook, without downloading or installing anything. Integrates seamlessly with the Wolfram Cloud.

- Clear and detailed documentation, with many examples, in both [Mathematica notebook](OGRe_Documentation.nb) and [PDF](OGRe_Documentation.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 continuous and active development. 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. Further-

more, 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 package 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

This package is compatible with Mathematica 12.0 or newer. It 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. It also ensures that you always use the latest version of the package, but be aware that updates may sometimes not be fully backwards compatible. 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"}]]`.
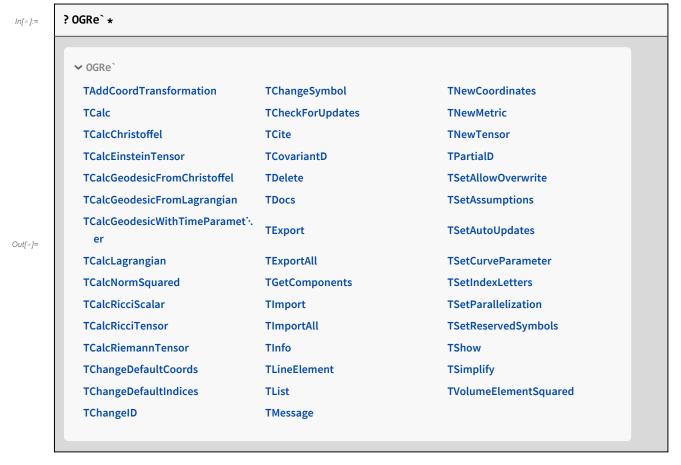
For the purposes of this documentation, I will use the "run from local file without installation" option, since you most likely downloaded both the documentation and the package together:

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

OGRe:

OGRe: An Object-Oriented General Relativity Package for Mathematica
By Barak Shoshany (baraksh@gmail.com) (baraksh.com)
v1.7.0 (2021-09-17)
GitHub repository: https://github.com/bshoshany/OGRe

- To view the full documentation for the package, type TDocs[].
- To list all available modules, type ?OGRe`*.
- To get help on a particular module, type **?** followed by the module name.
- To enable parallelization, type TSetParallelization[True].
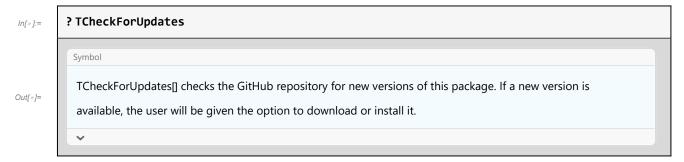- To check for updates, type TCheckForUpdates[].
  To enable automatic checks for updates at startup, type TSetAutoUpdates[True].

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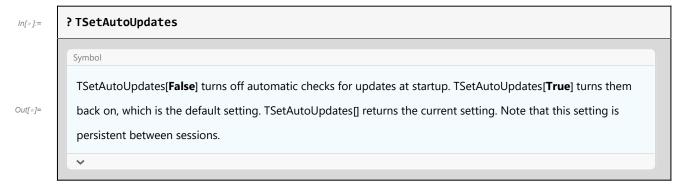
In[ ]:=
```
? OGRe`*
```

Out[ ]=

∨ OGRe`

TAddCoordTransformation	**TChangeSymbol**	**TNewCoordinates**
TCalc	**TCheckForUpdates**	**TNewMetric**
TCalcChristoffel	**TCite**	**TNewTensor**
TCalcEinsteinTensor	**TCovariantD**	**TPartialD**
TCalcGeodesicFromChristoffel	**TDelete**	**TSetAllowOverwrite**
TCalcGeodesicFromLagrangian	**TDocs**	**TSetAssumptions**
TCalcGeodesicWithTimeParamet⋱.er	**TExport**	**TSetAutoUpdates**
TCalcLagrangian	**TExportAll**	**TSetCurveParameter**
TCalcNormSquared	**TGetComponents**	**TSetIndexLetters**
TCalcRicciScalar	**TImport**	**TSetParallelization**
TCalcRicciTensor	**TImportAll**	**TSetReservedSymbols**
TCalcRiemannTensor	**TInfo**	**TShow**
TChangeDefaultCoords	**TLineElement**	**TSimplify**
TChangeDefaultIndices	**TList**	**TVolumeElementSquared**
TChangeID	**TMessage**	

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, whether built-in or from other packages.
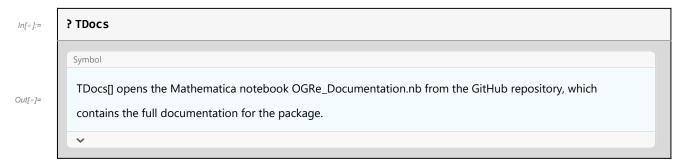
When the package loads, it will automatically check the GitHub repository for updates. This can also be done manually using the module `TCheckForUpdates`. If a new version of the package has been released, you will be given the option to download or install it in one click. If you are running the package directly from the web, you will always be using the latest version.

In[ ]:=

```
? TCheckForUpdates
```

Symbol

Out[ ]=

TCheckForUpdates[] checks the GitHub repository for new versions of this package. If a new version is available, the user will be given the option to download or install it.

⌄

You can also disable automatic checks for updates at startup in all future sessions by calling `TSetAutoUpdates[False]`:

In[ ]:=

```
? TSetAutoUpdates
```

Symbol

Out[ ]=

TSetAutoUpdates[**False**] turns off automatic checks for updates at startup. TSetAutoUpdates[**True**] turns them back on, which is the default setting. TSetAutoUpdates[] returns the current setting. Note that this setting is persistent between sessions.

⌄

If at any point you wish to view this documentation, simply type `TDocs[]`. This file (`OGRe_Documentation.nb`) will then be automatically opened in Mathematica.

In[ ]:=

```
? TDocs
```

Symbol

Out[ ]=

TDocs[] opens the Mathematica notebook OGRe_Documentation.nb from the GitHub repository, which contains the full documentation for the package.

⌄

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

package 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.

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

˅

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. Once a tensor object is created, you cannot create another tensor object with the same ID unless you rename or delete it first (see below). 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 through-out 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`. Further-more, if we, for example, want to take the derivative with respect to `r` (e.g. for the purpose of calculat-

ing 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 **TSetReservedSymbols** module:
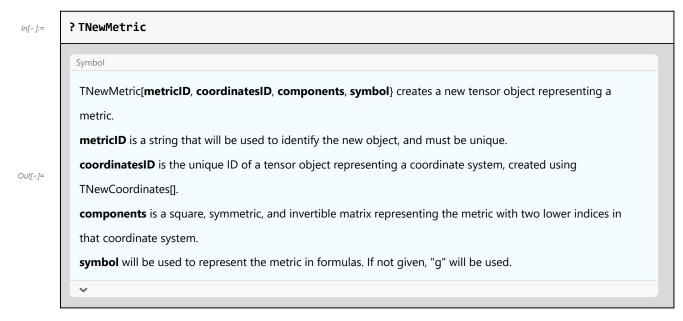
In[ ]:=

```
? TSetReservedSymbols
```

Symbol

TSetReservedSymbols[**symbol**] clears any definitions previously used for **symbol** and protects it against future changes. **symbol** can be either a symbol name or a string representing a symbol name. After completion, the module will return a list of the currently reserved symbols.

Useful for making sure coordinate variables, parameters, and abstract functions used in tensors remain abstract symbols and do not accidentally obtain values and break the code.

Out[ ]=

If the reserved symbol is a function, this function will be displayed without arguments when using TList[] and TShow[], for improved readability. The reserved symbols will be exported when using TExportAll[] so they can later be imported using TImportAll[].

TSetReservedSymbols[{**symbol1**, **symbol2**, …}] reserves all of the given symbols. Each of the symbols can be either a symbol name or a string representing a symbol name.

TSetReservedSymbols[] returns the currently reserved symbols.

⌄

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 **TSetReservedSymbols** 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 reserved symbols with numbers or other expressions using **/.** or **ReplaceAll**, so giving a value to a reserved symbol directly is not something you will ever need to do.

Defining metrics

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

In[ ]:= **? TNewMetric**

Symbol

Out[ ]=

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.

coordinatesID is the unique ID of a tensor object representing a coordinate system, created using TNewCoordinates[].

components is a square, symmetric, and invertible 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 reserved, that is, cleared and protected from future changes, using **TSetReservedSymbols**. However, **TNewMetric** does not automatically reserve any symbols it finds in the metric, since you might not want to reserve some of them. Since the Schwarzschild metric has a free parameter **M**, the mass of the black hole, we must reserve this symbol manually:

In[ ]:= **TSetReservedSymbols[M]**

Out[ ]= {t, x, y, z, r, θ, ϕ, M}

Note that **TSetReservedSymbols** returned the full list of symbols we reserved so far: the coordinate

symbols and the black hole's mass.

Now we can define the Schwarzschild metric:

In[●]:=
```
TNewMetric["Schwarzschild", "Spherical",
  DiagonalMatrix[{-(1 - (2 M)/r), 1/(1 - (2 M)/r), r², r² 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**] shows the components of the tensor object **ID** in its default index configuration and coordinate system.

TShow[**ID**, **indices**] shows the components in the index configuration **indices**, which should be a list of the form {±1, ±1, …}, where +1 corresponds to an upper index and −1 corresponds to a lower index.

TShow[**ID**, **coordinatesID**] shows the components in the coordinate system **coordinatesID**.

TShow[**ID**, **indices**, **coordinatesID**] shows the components in the index configuration **indices** and the coordinate system **coordinatesID**.

TShow[**ID**, **function**] maps **function** to each of the tensor's elements, and then automatically simplifies them, before they are displayed. Typically this would be ReplaceAll[rules] to apply the rules to the elements, but any function can be used.

TShow[**ID**, **indices**, **coordinatesID**, **function**] does all of the above; either **indices** or **coordinatesID** can be omitted.

⌄

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

In[ ]:= `TShow["Cartesian"]`

OGRe:

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

In[ ]:= `TShow["Spherical"]`

OGRe:

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

Note that coordinate tensors always have the symbol x.

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

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

OGRe:

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

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

OGRe:

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

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
```

Symbol

TList[**ID**] lists the unique, non-zero components of the tensor object **ID** in its default index configuration and coordinate system.

TList[**ID**, **indices**] lists the components in the index configuration **indices**, which should be a list of the form {±1, ±1, …}, where +1 corresponds to an upper index and −1 corresponds to a lower index.

TList[**ID**, **coordinatesID**] lists the components in the coordinate system **coordinatesID**.

Out[ ]=

TList[**ID**, **indices**, **coordinatesID**] lists the components in the index configuration **indices** and the coordinate system **coordinatesID**.

TList[**ID**, **function**] maps **function** to each of the tensor's elements, and then automatically simplifies them, before they are displayed. Typically this would be ReplaceAll[rules] to apply the rules to the elements, but any function can be used.

TList[**ID**, **indices**, **coordinatesID**, **function**] does all of the above; either **indices** or **coordinatesID** can be omitted.

⌄

For example:

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

OGRe:

$$\text{Minkowski:}$$
$$\eta_{tt} = -\eta_{xx} = -\eta_{yy} = -\eta_{zz} \quad = \quad -1$$

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

OGRe:

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

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.

Line and volume elements

In the case of metrics, we can also display them as a line element using the module `TLineElement`:

In[ ]:=
```
? TLineElement
```

Out[ ]=

> Symbol
>
> TLineElement[**metricID**] displays the line element of the metric **metricID** in its default coordinate system.
>
> TLineElement[**metricID**, **coordinatesID**] displays the line element in the coordinate system **coordinatesID**.
>
> ⌄

For example:

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

Out[ ]=
$$-dt^2 + dx^2 + dy^2 + dz^2$$

In[ ]:=
```
TLineElement["Schwarzschild"]
```

Out[ ]=
$$\frac{dr^2}{1 - \frac{2M}{r}} + \left(-1 + \frac{2M}{r}\right) dt^2 + r^2 d\theta^2 + r^2 d\phi^2 \sin[\theta]^2$$

Note that these are standard Mathematica expressions, so they can be manipulated like any other expressions, including operations such as simplifying or factoring. As an example of a more interesting (non-diagonal) line element, consider the **Alcubierre warp drive metric**:

In[ ]:=
```
TSetReservedSymbols[{v, f}];
TNewMetric["Alcubierre", "Cartesian",
   {{-1 + v[t]^2 f[t, x, y, z]^2, 0, 0, -v[t] × f[t, x, y, z]},
    {0, 1, 0, 0}, {0, 0, 1, 0}, {-v[t] × f[t, x, y, z], 0, 0, 1}}];
```

This is how the metric looks like in matrix form:

In[ ]:=
```
TShow["Alcubierre"]
```

OGRe:

$$\text{Alcubierre:}\quad g_{\mu\nu}(t, x, y, z) = \begin{pmatrix} -1 + f^2\, v^2 & 0 & 0 & -f\, v \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -f\, v & 0 & 0 & 1 \end{pmatrix}$$

f is a **form function** which is equal to 1 inside a "warp bubble" of finite radius and 0 outside it, and *v* is the velocity of the bubble, which can be faster than the speed of light ($v > 1$). Note that we reserved both *v* and *f*, since they are parameters used in the metric. Here we see another advantage of reserving symbols: since `v` and `f` are reserved symbols, and they are functions of the coordinates only, their arguments are not shown when using `TShow` and `TList`, for improved readability.

It is easy to see that the metric is flat where $f = 0$, that is, outside the bubble. Its line element is:

In[ ]:=
```
lineElement = TLineElement["Alcubierre"]
```

Out[ ]=
$$-\mathrm{d}t^2 + \mathrm{d}x^2 + \mathrm{d}y^2 + \mathrm{d}z^2 + \mathrm{d}t\, f[t, x, y, z] \times v[t]\ (-2\,\mathrm{d}z + \mathrm{d}t\, f[t, x, y, z] \times v[t])$$

We can simplify it by doing some clever factorization:

In[ ]:=
```
lineElement = lineElement // Expand // (#〚1〛 + #〚2〛 + #〚3〛 + Factor[#〚4〛 + #〚5〛 + #〚6〛]) &
```

Out[ ]=
$$-\mathrm{d}t^2 + \mathrm{d}x^2 + \mathrm{d}y^2 + (\mathrm{d}z - \mathrm{d}t\, f[t, x, y, z] \times v[t])^2$$

In this form, it is immediately clear that the metric is flat outside the warp bubble (where *f* is 0), and inside the warp bubble (when *f* is 1) it is a flat metric translated by an amount v[t] dt in the z direction:

In[ ]:=
```
lineElement /. f → (0 &)
```

Out[ ]=
$$-\mathrm{d}t^2 + \mathrm{d}x^2 + \mathrm{d}y^2 + \mathrm{d}z^2$$

In[ ]:=
```
lineElement /. f → (1 &)
```

Out[ ]=
$$-\mathrm{d}t^2 + \mathrm{d}x^2 + \mathrm{d}y^2 + (\mathrm{d}z - \mathrm{d}t\, v[t])^2$$

Another thing we can do with a metric is calculate its **volume elements squared**, which is simply the determinant of the metric:

In[ ]:= **? TVolumeElementSquared**

Out[ ]=

> Symbol
>
> TVolumeElement[**metricID**, **coordinatesID**] returns the determinant of the metric **metricID** in the coordinate system **coordinatesID**. If **coordinatesID** is not specified, the default coordinate system of the metric will be used. The square root of the determinant (or its negative, for a pseudo–Riemannian metric) is the volume element.
>
> ⌄

For example:

In[ ]:= **TVolumeElementSquared["Minkowski"]**

Out[ ]= -1

In[ ]:= **TVolumeElementSquared["Schwarzschild"]**

Out[ ]= $-r^4 \, \text{Sin}[\theta]^2$

In[ ]:= **TVolumeElementSquared["Alcubierre"]**

Out[ ]= -1

Choosing index letters

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

In[ ]:= **? TSetIndexLetters**

Out[ ]=

> Symbol
>
> TSetIndexLetters[] shows the index letters used when displaying indices.
>
> TSetIndexLetters[**letters**] changes the index letters.
>
> TSetIndexLetters[Automatic] resets the index letters to the default: "$\mu\nu\rho\sigma\kappa\lambda\alpha\beta\gamma\delta\epsilon\zeta\epsilon\theta\iota\xi\pi\tau\phi\chi\psi\omega$".
>
> ⌄

The default letters are:

In[ ]:= **TSetIndexLetters[]**

Out[ ]= $\mu\nu\rho\sigma\kappa\lambda\alpha\beta\gamma\delta\epsilon\zeta\epsilon\theta\iota\xi\pi\tau\phi\chi\psi\omega$

This means that the letter μ will be used for the first index, v 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[ ]:=   `TSetIndexLetters["abcdefghijklmnopqrstuvwxyz"]`

Out[ ]=   abcdefghijklmnopqrstuvwxyz

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

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

OGRe:

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

Finally, let us reset the letters to the default:

In[ ]:=   `TSetIndexLetters[Automatic]`

Out[ ]= $\mu\nu\rho\sigma\kappa\lambda\alpha\beta\gamma\delta\epsilon\zeta\epsilon\theta\iota\xi\pi\tau\phi\chi\psi\omega$

Note that `TList` always uses the coordinate symbols themselves for the indices (e.g. η_{tt}, η_{xx}, etc.), so it is not affected by `TSetIndexLetters`.

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**, **coordinatesID**, **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.
>
> **metricID** is the unique ID of a tensor object representing a metric, created using TNewMetric[]. The metric will be used to raise and lower indices for the new tensor.
>
> **coordinatesID** is the unique ID of a tensor object representing a coordinate system, created using TNewCoordinates[]. This coordinate system will be used to specify the components of the new tensor. If omitted, the default coordinate system of the metric **metricID** will be used.
>
> **indices** must be a list of the form {±1, ±1, …}, where +1 corresponds to an upper index and –1 corresponds to a lower index.
>
> **components** is a list specifying the representation of the tensor with the index configuration **indices** and in the coordinate system **coordinatesID**.
>
> **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 tensors from different manifolds together.

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 coordinate system. If a coordinate system is not specified, the default coordinate system of the associated metric will be used - but it is recommended to always specify the coordinate system explicitly, to avoid accidentally defining the tensor with the wrong components. 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", "Spherical", {}, {48 M²/r⁶}, "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"]
```

OGRe:

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

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. (We do not need to reserve the symbol v, since we already reserved it for the Alcubierre metric above.)

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", "Cartesian", {1}, {1, v, 0, 0}/√(1 - v²)]
```

OGRe:

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

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 reserve the symbols ρ (for the energy density) and **p** (for the pressure):

In[ ]:=
```
TSetReservedSymbols[{ρ, p}]
```

Out[ ]=
```
{t, x, y, z, r, θ, φ, M, v, f, 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",
    "Cartesian", {1, 1}, DiagonalMatrix[{ρ, p, p, p}], "T"]
```

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

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"]
```

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

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

references to **oldID** will be replaced with **newID**.

⌄

For example, let us change the ID of the 4-velocity tensor from **"FourVelocity"** to **"4‑Velocity"**:

In[ ]:=
```
TChangeID["FourVelocity" → "4‑Velocity"]
```

Out[ ]=
```
4‑Velocity
```

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

In[ ]:=
```
TShow["FourVelocity"]
```

TMessage: The tensor "FourVelocity" does not exist.

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

Note that this error is associated with the symbol **TMessage**. Any message not associated with a specific OGRe module will be associated with this symbol.

We can access the tensor using the new ID:

In[ ]:=
```
TShow["4‑Velocity"]
```

OGRe:

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

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.

Deleting and overwriting tensors

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

In[ ]:=
```
? TDelete
```

Out[ ]=

Symbol

TDelete[**ID**] permanently deletes the tensor object **ID**. If the tensor is a metric or coordinate system, it cannot

be deleted unless all tensors referring to it have been deleted 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
```

There is no module to change the components of a tensor after it has already been defined, as this may break class invariants (in other words, introduce inconsistencies in the data). Instead, you must create a new tensor with the same ID using **TNewTensor**. By default, OGRe does not allow overwriting tensors, to prevent loss of data:

In[ ]:=
$$\text{TNewTensor}\left["4-Velocity", "Minkowski", "Cartesian", \{1\}, \frac{\{1, v, 0, 0\}}{\sqrt{1 - v^2}}\right]$$

TMessage: Overwriting the tensor "4–Velocity".

Out[ ]=
```
4-Velocity
```

Users who want to be able to create new tensors with the same ID as an existing tensor without deleting the old tensor first, and are confident that they will not accidentally lose any data by doing so, may

enable overwriting tensors using **TSetAllowOverwrite**:

In[ ]:=
```
? TSetAllowOverwrite
```

Out[ ]=

Symbol

TSetAllowOverwrite[**True**] allows overwriting tensors. If the user creates a new tensor with the same ID as an existing tensor, the latter will be overwritten. Note that this can result in loss of data.

TSetAllowOverwrite[**False**] disallows overwriting, which is the default setting. TSetAllowOverwrite[] returns the current setting. Note that this setting is persistent between sessions.

⌄

First let us save the current state of **TSetAllowOverwrite**:

In[ ]:=
```
currentOverwrite = TSetAllowOverwrite[]
```

Out[ ]=
```
True
```

Let us now turn it on:

In[ ]:=
```
TSetAllowOverwrite[True]
```

TSetAllowOverwrite: Overwriting tensors turned on.

We can now overwrite the tensor "4-Velocity" by creating a new tensor with the same ID:

In[ ]:=
$$\text{TNewTensor}\left[\text{"4-Velocity", "Minkowski", "Cartesian", \{1\}, } \frac{\{1, v, 0, 0\}}{\sqrt{1 - v^2}}\right]$$

TMessage: Overwriting the tensor "4-Velocity".

Out[ ]=
```
4-Velocity
```

Like any other message, the message warning us that we are overwriting a tensor can be turned off using **Off**. In this case, the command to turn it if would be **Off[TMessage::WarningOverwrite]**.

Let us switch overwriting tensors back to the previous setting:

In[ ]:=
```
TSetAllowOverwrite[currentOverwrite]
```

TSetAllowOverwrite: Overwriting tensors turned on.

Note that this setting is persistent between sessions - if you turn overwriting on, then it will remain on permanently, even in other Mathematica sessions, until you turn it back off.

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, $v^\vee$, 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 $v \in \{0, 1, 2, 3\}$:

$$V_\mu = \sum_{\nu=0}^{3} g_{\mu\nu} \, V^\nu = g_{\mu 0} \, V^0 + g_{\mu 1} \, V^1 + g_{\mu 2} \, V^2 + g_{\mu 3} \, V^3 \, .$$

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}]
```

OGRe:

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

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}]`

OGRe:

$$\text{PerfectFluid:}$$

$$T^t_{\ t} \qquad\qquad = \ -\rho$$
$$T^x_{\ x} = T^y_{\ y} = T^z_{\ z} \ = \ p$$

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

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

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

In[ ]:= `TShow["Schwarzschild", {1, 1}]`

OGRe:

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

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

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

OGRe:

$$\text{Schwarzschild:}$$

$$g^t_{\ t} = g^r_{\ r} = g^\theta_{\ \theta} = g^\phi_{\ \phi} \ = \ 1$$

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
```

Symbol

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

Out[ ]=

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

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

In[ ]:=

```
TChangeDefaultIndices["PerfectFluid", {-1, -1}]
```

Out[ ]=

```
PerfectFluid
```

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

In[ ]:=

```
TShow["PerfectFluid"]
```

OGRe:

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

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:

$$\text{T}_{\alpha'\beta'}\left(\text{x}^{\mu'}\right) = \frac{\partial x^\alpha}{\partial x^{\alpha'}} \frac{\partial x^\beta}{\partial x^{\beta'}} \text{T}_{\alpha\beta}\left(\text{x}^\mu\right).$$

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

$$
T^{\alpha_1' \cdots \alpha_p'}_{\beta_1' \cdots \beta_q'} \left(\mathbf{x}^{\mu'} \right) = \left(\frac{\partial \mathbf{x}^{\alpha_1'}}{\partial \mathbf{x}^{\alpha_1}} \cdots \frac{\partial \mathbf{x}^{\alpha_p'}}{\partial \mathbf{x}^{\alpha_p}} \right) \left(\frac{\partial \mathbf{x}^{\beta_1}}{\partial \mathbf{x}^{\beta_1'}} \cdots \frac{\partial \mathbf{x}^{\beta_q}}{\partial \mathbf{x}^{\beta_q'}} \right) T^{\alpha_1 \cdots \alpha_p}_{\beta_1 \cdots \beta_q} \left(\mathbf{x}^{\mu} \right) .
$$

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 $\leftrightarrow$ 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 denominator, 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**.

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[●]:= $\texttt{TAddCoordTransformation}\Big[\texttt{"Spherical"} \to \texttt{"Cartesian"},$

$\Big\{ r \to \sqrt{x^2 + y^2 + z^2} ,\ \theta \to \texttt{ArcCos}\Big[\dfrac{z}{\sqrt{x^2 + y^2 + z^2}} \Big],\ \phi \to \texttt{ArcTan}[x, y] \Big\}\Big];$

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"]
```

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

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

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

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

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

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

OGRe:
$$\text{Kretschmann:}$$
$$K \quad = \quad \frac{48 \, M^2}{\left(x^2 + y^2 + z^2\right)^3}$$

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"]`

Out[ ]=   `PerfectFluid`

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"]`

OGRe:

<div align="center">

PerfectFluid:

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

</div>

Applying a function to the tensor components

Optionally, a function can be provided to **TList** and **TShow** as their last argument. If provided, the function will be applied to each of the tensor's components, the components will be simplified, and the tensor will be displayed with the new components. Note that this only applies to displaying the components; the tensor data itself will not change.

Typically, the applied function will be **ReplaceAll** to replace a particular symbol in the tensor with another symbol or numerical value. For example, perhaps we would like to display the value of the Kretschmann scalar for a particular choice of M and r:

In[ ]:=   `TShow["Kretschmann", ReplaceAll[{M → 1, r → 10}]]`

OGRe:

<div align="center">

Kretschmann: $K\,(t, r, \theta, \phi) = \dfrac{3}{62\,500}$

</div>

Or maybe we would like to display the perfect fluid stress tensor with p equal to ρ:

In[ ]:=
```
TList["PerfectFluid", ReplaceAll[p → ρ]]
```

OGRe:

$$\text{PerfectFluid:}$$

$$\mathsf{T}_{tt} = \mathsf{T}_{rr} \;\; = \;\; \rho$$

$$\mathsf{T}_{\theta\theta} \;\;\;\;\;\; = \;\; r^2\,\rho$$

$$\mathsf{T}_{\phi\phi} \;\;\;\;\;\; = \;\; r^2\,\rho\,\mathsf{Sin}[\theta]^2$$

The function can, of course, also be combined with a choice of indices and/or coordinates:

In[ ]:=
```
TList["PerfectFluid", {1, 1}, "Cartesian", ReplaceAll[p → ρ]]
```

OGRe:

$$\text{PerfectFluid:}$$

$$\mathsf{T}^{tt} = \mathsf{T}^{xx} = \mathsf{T}^{yy} = \mathsf{T}^{zz} \;\; = \;\; \rho$$

There are other useful functions we can apply besides `ReplaceAll`. We will see an example below.

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", "Cartesian", {}, { √(x² + y² + z²) }, "d"]
```

OGRe:

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

When we convert this scalar to spherical coordinates, we expect to get r, but instead we get the absolute value of r:

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

OGRe:

$$\text{SpatialDistance:}\;\; \mathsf{d}\,(t, r, \theta, \phi) = \mathsf{Abs}[r]$$

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. By default, OGRe uses the simplification assumption that all variables (coordinates, parameters, etc.) are **real**. Therefore, Mathematica knows to simplify $\sqrt{r^2}$ to $|\,r\,|$. However, Mathematica doesn't automatically know that r is non-negative.

The user may specify which assumptions to pass to `FullSimplify` using the module `TSetAssumptions`:

In[ ]:=
```
? TSetAssumptions
```

Out[ ]=

Symbol

TSetAssumptions[] shows the assumptions to be used when simplifying expressions.

TSetAssumptions[**assumptions**] appends new assumptions to the previously added assumptions.

TSetAssumptions[None] clears all previously added assumptions.

TSetAssumptions[!Reals] disables the default assumption that all variable are real, which secretly adds the assumption Element[_, Reals] to the list of assumptions. TSetAssumptions[Reals] re-enables this assumption. The output of this module is always an Association indicating whether variables are assumed to be real and listing the user-defined 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[ ]:=
```
TSetAssumptions[r ≥ 0]
```

Out[ ]=
$$\langle | \text{AssumeReal} \to \text{True}, \text{User} \to \{r \geq 0\} | \rangle$$

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"]
```

OGRe:

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

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 automatically, 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−
>
> defined simplification assumptions set using TSetAssumptions[]. To be used if the assumptions have changed
>
> after the components have already been calculated.
>
> TSimplify[**expression**] simplifies **expression** based on the user−defined simplification assumptions. If
>
> **expression** is a list, the components will be simplified in parallel.
>
> ⌄

Now we get the expected result:

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

OGRe:

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

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 `@`.

Finally, we note that if you are using non-real variables, you can disable the assumption that all variables are real using `TSetAssumptions[!Reals]`. If you later want to turn it back on, use TSetAssumptions`[Reals]`. The value of the key **"AssumeReal"** in the output of `TSetAssumptions` indicates whether this assumption is turned on.

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[ ]=

$$\left\langle \left| \, 4\text{-Velocity} \to \left\langle \left| \, \text{Components} \to \left\langle \left| \, \{\{1\}, \text{Cartesian}\} \to \left\{ \frac{1}{\sqrt{1-v^2}}, \frac{v}{\sqrt{1-v^2}}, 0, 0 \right\}, \right. \right. \right. \right. \right.$$

$$\left. \{\{-1\}, \text{Cartesian}\} \to \left\{ -\frac{1}{\sqrt{1-v^2}}, \frac{v}{\sqrt{1-v^2}}, 0, 0 \right\} \left| \right\rangle, \right.$$

$$\text{DefaultCoords} \to \text{Cartesian}, \text{DefaultIndices} \to \{1\}, \text{Metric} \to \text{Minkowski},$$

$$\left. \text{Role} \to \text{Tensor}, \text{Symbol} \to \square, \text{OGReVersion} \to \text{v1.7.0} \ (2021\text{-}09\text{-}17) \left| \right\rangle \left| \right\rangle \right.$$

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, which depends on the module that created it. Will be **"Coordinates"** if the tensor was created using **TNewCoordinates**, **"Metric"** if the tensor was created using **TNewMetric**, or **"Tensor"** if the tensor was created using **TNewTensor**. Other modules that we will discuss below, such as **TCalc**, **TCalcChristoffel**, and **TCalcRiemannTensor**, have corresponding roles as well. Additional roles are only 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.

- **"OGReVersion"**: The version of the package used to create the tensor. (Note that this key is not stored internally, it is added by **TExport**.)

Other keys also exist in special cases. For example, for tensor objects representing coordinate systems,

the keys **"CoordTransformations"** and **"Jacobians"** are used to store the details of 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]} |⟩,
    Jacobians → ⟨| Spherical → ⟨| Jacobian →
        {{1, 0, 0, 0}, {0, Cos[ϕ] Sin[θ], r Cos[θ] Cos[ϕ], -r Sin[θ] Sin[ϕ]}, {0,
          Sin[θ] Sin[ϕ], r Cos[θ] Sin[ϕ], r Cos[ϕ] Sin[θ]}, {0, Cos[θ], -r Sin[θ], 0}},
      InverseJacobian → {{1, 0, 0, 0}, {0, Cos[ϕ] Sin[θ], Sin[θ] Sin[ϕ], Cos[θ]},
        {0, $\frac{\text{Cos}[θ]\,\text{Cos}[ϕ]}{r}$, $\frac{\text{Cos}[θ]\,\text{Sin}[ϕ]}{r}$, $-\frac{\text{Sin}[θ]}{r}$},
        {0, $-\frac{\text{Csc}[θ]\,\text{Sin}[ϕ]}{r}$, $\frac{\text{Cos}[ϕ]\,\text{Csc}[θ]}{r}$, 0}}, ChristoffelJacobian →
      {{{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}}, {{0, 0, 0, 0},
        {0, 0, Cos[θ] Cos[ϕ], -Sin[θ] Sin[ϕ]}, {0, Cos[θ] Cos[ϕ], -r Cos[ϕ] Sin[θ],
          -r Cos[θ] Sin[ϕ]}, {0, -Sin[θ] Sin[ϕ], -r Cos[θ] Sin[ϕ], -r Cos[ϕ] Sin[θ]}},
      {{0, 0, 0, 0}, {0, 0, Cos[θ] Sin[ϕ], Cos[ϕ] Sin[θ]},
        {0, Cos[θ] Sin[ϕ], -r Sin[θ] Sin[ϕ], r Cos[θ] Cos[ϕ]},
        {0, Cos[ϕ] Sin[θ], r Cos[θ] Cos[ϕ], -r Sin[θ] Sin[ϕ]}}, {{0, 0, 0, 0},
        {0, 0, -Sin[θ], 0}, {0, -Sin[θ], -r Cos[θ], 0}, {0, 0, 0, 0}}} |⟩ |⟩,
    OGReVersion → v1.7.0 (2021-09-17) |⟩ |⟩

The output of **TExport** is a raw **Association**, and is not intended to be read by humans. If you wish to display the information encoded in the object in human-readable form, use **TInfo**:

In[ ]:=
```
? TInfo
```

Out[ ]=

Symbol

TInfo[] lists all the tensors created so far in this session: coordinate systems, metrics, and the tensors associated with each metric.

TInfo[**ID**] displays information about the tensor object **ID**, including its symbol, role, associated metric, and default coordinates and indices, in human–readable form.

If **ID** represents a coordinate system, displays a list of all tensors using it as their default coordinate system.

If **ID** represents a metric, displays a list of all tensors using it as their associated metric.

⌄

TInfo will also tell you which other tensor objects use this tensor as their metric or default coordinate system, if applicable. For example:

In[●]:=

```
TInfo["Cartesian"]
```

OGRe:

ID: Cartesian
Symbol: x
Role: Coordinates
Default Indices: {1}
Default Coordinates For: Minkowski , Alcubierre , 4-Velocity , SpatialDistance
Components: TList | TShow

In[●]:=

```
TInfo["Minkowski"]
```

OGRe:

ID: Minkowski
Symbol: η
Role: Metric
Default Coordinates: Cartesian
Default Indices: {−1, −1}
Tensors Using This Metric: PerfectFluid , 4-Velocity , SpatialDistance
Components: TList | TShow

In[●]:=

```
TInfo["PerfectFluid"]
```

OGRe:

ID: PerfectFluid
Symbol: T
Role: Tensor
Metric: Minkowski
Default Coordinates: Spherical
Default Indices: {−1, −1}
Components: TList | TShow

You can click on the links to **TList** and **TShow** at the bottom to display the components of the tensor being inspected. You can also click on any tensor name in the output (e.g. **Minkowski**, **4-Velocity**, and **SpatialDistance** in **TInfo["Cartesian"]**) to execute **TInfo** for that tensor. In addition, calling **TInfo[]** with no parameters lists all the tensors created so far in this session - coordinate systems, metrics, and the tensors associated with each metric:

In[ ]:=

TInfo[]

OGRe:

Total tensors created: 9
Coordinate Systems:
1. Cartesian
2. Spherical
Metrics:
1. **Alcubierre** →
2. **Minkowski** → PerfectFluid │ 4-Velocity │ SpatialDistance
3. **Schwarzschild** → Kretschmann

We see that we created 9 tensors in total so far: 2 coordinate systems, 3 metrics, 3 tensors associated with the Minkowski metric, and 1 tensor associated with the Schwarzschild metric.

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[].

⌄

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

In[ ]:=

? TExportAll

Out[ ]=

Symbol

TExportAll[] exports the raw tensor data for all tensors defined in the current session as an Association.
TExportAll[**filename**] exports the data to **filename**. If a full path is not given, the file will be created in the current working directory, as given by Directory[]. This directory can be changed using SetDirectory[]. Note that the file will be overwritten if 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[ ]=

{Options, Cartesian, Spherical, Minkowski, Schwarzschild,
  Alcubierre, Kretschmann, PerfectFluid, 4-Velocity, SpatialDistance}

TExportAll exports not only the tensors, but also a special key called **Options**, which contains the current version of the package (for compatibility purposes, in case the storage format changes in future versions) and any options set by the user during the session. Note that keys associated with tensor objects, such as **"Cartesian"**, **"Spherical"**, and so on, are always strings, but **Options** is not a string; this is to ensure it doesn't get accidentally interpreted as a tensor object. We can see the options configured in this session by reading the value of the **Options** key:

In[ ]:=
```
TExportAll[][Options]
```

Out[ ]=
```
⟨| CurveParameter → Global`λ, IndexLetters → μνρσκλαβγδεζϵθιξπτϕχψω,
   OGReVersion → v1.7.0 (2021-09-17), Parallelize → False,
   ReservedSymbols → {t, x, y, z, r, θ, ϕ, M, v, f, ρ, p},
   SimplifyAssumptions → ⟨| AssumeReal → True, User → {r ≥ 0} |⟩ |⟩
```

Note that several settings we have already discussed, such as the index letters, reserved symbols, and simplification assumptions, are included. This means that these settings will be imported when you import the tensor data in another Mathematica session - you can start right where you left off, no need to reconfigure anything.

The output of **TExportAll** 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 is assumed to be located in the current working directory, as given by Directory[]. This directory can be changed using SetDirectory[].

WARNING: 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**. Similarly, any settings configured during the session will be replaced with the imported settings.

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"]**.

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**!

Finally, sometimes you may want to extract the components of a tensor in a specific representation as a `List` so you can use them outside of this package, as regular Mathematica expressions rather than tensor objects. This is done using `TGetComponents`:

In[ ]:=
```
? TGetComponents
```

Symbol

Out[ ]=

TGetComponents[**ID**, **indices**, **coordinatesID**] extracts the components of the tensor object **ID** with the index

configuration **indices** and in the coordinate system **coordinatesID** as a list.

indices must be a list of the form {±1, ±1, …}, where +1 corresponds to an upper index and –1 corresponds to

a lower index.

If **indices** and/or **coordinatesID** are omitted, the default values are used, and a message will let the user

know which representation the components are given in, to avoid confusion.

TGetComponents[**ID**, **indices**, **coordinatesID**, **function**] maps **function** to each of the tensor's elements, and

then automatically simplifies them, before they are returned. Typically this would be ReplaceAll[rules] to apply

the rules to the elements, but any function can be used.

For example :

In[ ]:=
```
InverseSchwarzschild = TGetComponents["Schwarzschild", {1, 1}, "Spherical"]
```

Out[ ]=
$$\left\{ \left\{ \frac{r}{2\,M-r}, 0, 0, 0 \right\}, \left\{ 0, 1 - \frac{2\,M}{r}, 0, 0 \right\}, \left\{ 0, 0, \frac{1}{r^2}, 0 \right\}, \left\{ 0, 0, 0, \frac{Csc[\theta]^2}{r^2} \right\} \right\}$$

We can now treat `InverseSchwarzschild` as any other `List` in Mathematica - for example, extract the element at a particular position:

In[ ]:=
```
InverseSchwarzschild[[1, 1]]
```

Out[ ]=
$$\frac{r}{2\,M-r}$$

If the desired index configuration and/or coordinate system are not specified, the default ones will be used. However, it is important to always know exactly which representation the components are in, to avoid confusion. Thus, you will be notified which representation was used:

In[ ]:= `TGetComponents["Schwarzschild"]`

TGetComponents: Using the default index configuration {–1, –1} and the default coordinate system "Spherical".

Out[ ]= $\left\{\left\{-1 + \dfrac{2\,M}{r}, 0, 0, 0\right\}, \left\{0, \dfrac{1}{1 - \frac{2\,M}{r}}, 0, 0\right\}, \left\{0, 0, r^2, 0\right\}, \left\{0, 0, 0, r^2 \operatorname{Sin}[\theta]^2\right\}\right\}$

TGetComponents can take as its last argument a function, which will be applied to the elements. The elements will then be simplified, and the result will be returned. This works exactly the same as providing a function to **TList** and **TShow** (see above). For example, here are the components of the Schwarzschild metric on the hypersurface with $\theta = \pi/2$:

In[ ]:= `TGetComponents["Schwarzschild", ReplaceAll[θ → $\dfrac{\pi}{2}$]]`

TGetComponents: Using the default index configuration {–1, –1} and the default coordinate system "Spherical".

Out[ ]= $\left\{\left\{-1 + \dfrac{2\,M}{r}, 0, 0, 0\right\}, \left\{0, \dfrac{1}{1 - \frac{2\,M}{r}}, 0, 0\right\}, \left\{0, 0, r^2, 0\right\}, \left\{0, 0, 0, r^2\right\}\right\}$

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[**formula**] calculates a tensor **formula**, which may involve 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"["$\mu\nu$"].

TCalc[**targetID**[**targetIndices**], **formula**, **symbol**] calculates a tensor **formula** and stores the result in a new tensor object.

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

targetIndices specifies the order of indices of the resulting tensor. The indices must be a permutation of the free indices of **formula**. If omitted, the indices are assumed to be in the same order as they appear in **formula**.

symbol specifies the symbol to use for the resulting tensor. 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, **"$\alpha\beta\gamma$"**, **"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"]]
```

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

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"]
```

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

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"]
```

OGRe:

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

(We deleted the result to avoid cluttering memory 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", "Cartesian",
  {-1, -1}, {{0, 0, 0, 1}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}}, "N"]
```

OGRe:

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

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

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

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

OGRe:

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

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

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

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

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

Since the order of 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:

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

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

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

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:

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

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

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

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:

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

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

In[ ]:=

```
TShow@TCalc[2 "Minkowski"["μν"]]
```

OGRe:

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

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:

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

In[ ]:=

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

OGRe:

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

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:

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

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:

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

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["PerfectFluidFromVelocity",
  (ρ + p) "4-Velocity"["μ"]."4-Velocity"["ν"] + p "Minkowski"["μν"], "T"]
```

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

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

In[ ]:=
```
TShow["PerfectFluidFromVelocity", ReplaceAll[v → 0]]
```

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

Note that here, the second argument is a function to apply to the tensor elements. Typically this would be **ReplaceAll[rules]** to apply the **rules** to the elements, as we did here with the rule **v→0**. Another example of a potential use is **Map[function]** to map a specific function to each element. This can be done with both **TShow** and **TList**.

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"]`

OGRe:

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

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"]`

OGRe:

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

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"]]`

OGRe:

$$\text{Result:}\quad \square\,(t, x, y, z) = -1$$

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

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

OGRe:

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

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"]]`

OGRe:

$$\text{Result:}\quad \square\,(t, x, y, z) = 4$$

In[◦]:= `TShow@TCalc["PerfectFluidFromVelocity"["μμ"]]`

OGRe:

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

Of course, this also works for tensors with more than two indices, as we will see below. Any combina-

tion 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 rank higher**. For example, we can calculate the gradient $\partial_\mu K$ of the Kretschmann scalar as follows:

In[ ]:=
```
TShow@TCalc[TPartialD["μ"]."Kretschmann"[""]]
```

OGRe:

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

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 rank lower**. For example, here is the divergence $\partial_\mu x^\mu$ of the spherical coordinate vector:

In[ ]:=
```
TShow@TCalc[TPartialD["μ"]."Spherical"["μ"]]
```

OGRe:

$$\text{Result: } \square\,(t, r, \theta, \phi) = 4$$

As you can see, the syntax for both the gradient and divergence 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.

WARNING: When applying partial derivatives to tensors, the result generally does not transform like a tensor under a coordinate transformation. For this reason, in general relativity we normally use the **covariant derivative** instead of a partial derivative. However, there are three important exceptions, where partial derivatives must be used: in the covariant derivative itself, the **Levi-Civita connec-**

tion, and the **Riemann tensor**, all of which will be discussed below.

Of these three special cases, the covariant derivative and the Riemann tensor turn out to nonetheless transform like tensors under coordinate transformations, due to cancellations. However, the Levi-Civita connection, whose components are called the **Christoffel symbols**, has a special transformation rule, which is used automatically by OGRe, as we will show.

In all other cases, if the user creates an arbitrary tensor using partial derivatives, the result will generally **transform incorrectly** under a coordinate transformation in OGRe. Therefore, it is highly recommended to avoid using partial derivatives with `TCalc` unless you really know what you're doing.

The Christoffel symbols

The **Christoffel symbols** are a very important tensor-like objects in differential geometry. They are the components of the **Levi-Civita connection,** which is the unique torsion-free connection that preserves the metric. 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"["`$\nu\sigma$`"]` 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[1/2 "Schwarzschild"["λσ"].

    (TPartialD["μ"]."Schwarzschild"["νσ"] + TPartialD["ν"]."Schwarzschild"["σμ"] -
      TPartialD["σ"]."Schwarzschild"["μν"]), "Γ"], {1, -1, -1}]
```

OGRe:

$$\text{Result:}$$

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

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

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

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

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

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

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

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

However, there is a problem; as we mentioned above, **the Christoffel symbols are not the components of a tensor**, meaning that the Levi-Civita connection does not transform as a tensor does under a coordinate transformation. Indeed, by transforming the metric in the definition, one can show that

$$\Gamma^{\lambda'}{}_{\mu'\,\nu'} = \frac{\partial\,\mathsf{x}^{\lambda'}}{\partial\,\mathsf{x}^\lambda}\,\frac{\partial\,\mathsf{x}^\mu}{\partial\,\mathsf{x}^{\mu'}}\,\frac{\partial\,\mathsf{x}^\nu}{\partial\,\mathsf{x}^{\nu'}}\,\Gamma^\lambda{}_{\mu\nu} + \frac{\partial\,\mathsf{x}^{\lambda'}}{\partial\,\mathsf{x}^\lambda}\,\frac{\partial^2\,\mathsf{x}^\lambda}{\partial\,\mathsf{x}^{\mu'}\,\partial\,\mathsf{x}^{\nu'}}\,.$$

The first term is the familiar transformation rule for a tensor, with one factor of the Jacobian per index as usual. However, there is also an extra second term, meaning that the Christoffel symbols do not transform like a tensor.

(Actually, you are also not supposed to raise or lower indices in the Christoffel symbols, but in practice, you can do that as long as you make it clear that it's just an abuse of notation - you are only adding factors of the metric, not creating a new tensor representation with different transformation properties.)

Due to the extra transformation term, the tensor object we calculated manually above using `TCalc` **must not be used**. Instead, we should use the built-in module TCalcChristoffel, which not only performs the calculation automatically for us, but also marks the result as a special object with special transformation properties:

In[●]:=   **? TCalcChristoffel**

Symbol

Out[●]=

TCalcChristoffel[**metricID**] calculates the Christoffel symbols (the coefficients of the Levi–Civita connection)

from the metric **metricID** and stores the result in a new tensor object with ID "**metricID**Christoffel". Note

that the Christoffel symbols are not the components of a tensor, but this tensor object will know to transform

according to the correct rules under change of coordinates.

Let us, then, calculate the Christoffel symbols for the Schwarzschild metric properly, using **TCalcChristoffel**:

In[●]:=   **TList@TCalcChristoffel["Schwarzschild"]**

OGRe:

$$\text{SchwarzschildChristoffel:}$$

$$\Gamma^t_{\ tr} = \Gamma^t_{\ rt} \qquad = \quad \frac{M}{r\,(-2\,M+r)}$$

$$\Gamma^r_{\ tt} \qquad = \quad \frac{M\,(-2\,M+r)}{r^3}$$

$$\Gamma^r_{\ rr} \qquad = \quad \frac{M}{2\,M\,r-r^2}$$

$$\Gamma^r_{\ \theta\theta} \qquad = \quad 2\,M-r$$

$$\Gamma^r_{\ \phi\phi} \qquad = \quad (2\,M-r)\,\text{Sin}[\theta]^2$$

$$\Gamma^\theta_{\ r\theta} = \Gamma^\theta_{\ \theta r} = \Gamma^\phi_{\ r\phi} = \Gamma^\phi_{\ \phi r} \quad = \quad \frac{1}{r}$$

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

$$\Gamma^\phi_{\ \theta\phi} = \Gamma^\phi_{\ \phi\theta} \qquad = \quad \text{Cot}[\theta]$$

These are the same components we got before, but now they will transform properly. Note that the name of the tensor object created by **TCalcChristoffel** is always the name of metric with the word **Christoffel** appended to it (no spaces).

For maximal clarity, let us demonstrate the discrepancy in the coordinate transformation with a simple test metric:

In[ ]:= `TShow@TNewMetric["SimpleMetric", "Cartesian", DiagonalMatrix[{-x, 1, 1, 1}]]`

OGRe:

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

We calculate its Christoffel symbols in two ways. First manually, as we did above for the Schwarzschild metric:

In[ ]:= `TList@TChangeDefaultIndices[`
`  TCalc["SimpleMetricManualChristoffel", ` $\frac{1}{2}$ `"SimpleMetric"["λσ"].`
`    (TPartialD["μ"]."SimpleMetric"["νσ"] + TPartialD["ν"]."SimpleMetric"["σμ"] -`
`      TPartialD["σ"]."SimpleMetric"["μν"]), "Γ"], {1, -1, -1}]`

OGRe:

SimpleMetricManualChristoffel:

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

Then, with the built-in module `TCalcChristoffel`:

In[ ]:= `TList@TCalcChristoffel["SimpleMetric"]`

OGRe:

SimpleMetricChristoffel:

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

The two results have the same components, as expected. But now, let us now transform them to spherical coordinates. First, we transform the tensor object obtained using `TCalcChristoffel`:

In[ ]:= `TList["SimpleMetricChristoffel", "Spherical"]`

OGRe:

$$\text{SimpleMetricChristoffel:}$$

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

$$\Gamma^t{}_{t\theta} = \Gamma^t{}_{\theta t} \qquad\qquad = \frac{\text{Cot}[\theta]}{2}$$

$$\Gamma^t{}_{t\phi} = \Gamma^t{}_{\phi t} \qquad\qquad = -\frac{\text{Tan}[\phi]}{2}$$

$$\Gamma^r{}_{tt} \qquad\qquad = \frac{1}{2}\,\text{Cos}[\phi]\,\text{Sin}[\theta]$$

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

$$\Gamma^r{}_{\phi\phi} \qquad\qquad = -r\,\text{Sin}[\theta]^2$$

$$\Gamma^\theta{}_{tt} \qquad\qquad = \frac{\text{Cos}[\theta]\,\text{Cos}[\phi]}{2r}$$

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

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

$$\Gamma^\phi{}_{tt} \qquad\qquad = -\frac{\text{Csc}[\theta]\,\text{Sin}[\phi]}{2r}$$

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

This is the **correct** representation of the Christoffel symbols in spherical coordinates, as the extra term in the transformation was taken into account. However, if we transform the Christoffel symbols we obtained manually using `TCalc`, we get:

In[ ]:= `TList["SimpleMetricManualChristoffel", "Spherical"]`

OGRe:

$$\text{SimpleMetricManualChristoffel:}$$

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

$$\Gamma^t{}_{t\theta} = \Gamma^t{}_{\theta t} \quad = \frac{\text{Cot}[\theta]}{2}$$

$$\Gamma^t{}_{t\phi} = \Gamma^t{}_{\phi t} \quad = -\frac{\text{Tan}[\phi]}{2}$$

$$\Gamma^r{}_{tt} \quad = \frac{1}{2}\,\text{Cos}[\phi]\,\text{Sin}[\theta]$$

$$\Gamma^\theta{}_{tt} \quad = \frac{\text{Cos}[\theta]\,\text{Cos}[\phi]}{2r}$$

$$\Gamma^\phi{}_{tt} \quad = -\frac{\text{Csc}[\theta]\,\text{Sin}[\phi]}{2r}$$

This is **not** the correct result, since the transformation did not take into account the extra term. To

verify that the former result is indeed the correct one, let us change the default coordinate system of `SimpleMetric` to spherical:

In[ ]:=
```
TChangeDefaultCoords["SimpleMetric", "Spherical"];
```

Now, when we calculate the Christoffel symbols manually from this metric, we will get their correct representation in spherical coordinates. This is because `TCalc` always performs the calculations internally in the default coordinates of the first tensor, so the result was calculated **from scratch** in spherical coordinates, instead of being calculated first in Cartesian coordinates and then transformed:

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

OGRe:

$$\text{SimpleMetricManualChristoffelSpherical:}$$

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

$$\Gamma^t{}_{t\theta} = \Gamma^t{}_{\theta t} \qquad = \quad \frac{\text{Cot}[\theta]}{2}$$

$$\Gamma^t{}_{t\phi} = \Gamma^t{}_{\phi t} \qquad = \quad -\frac{\text{Tan}[\phi]}{2}$$

$$\Gamma^r{}_{tt} \qquad = \quad \frac{1}{2}\,\text{Cos}[\phi]\,\text{Sin}[\theta]$$

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

$$\Gamma^r{}_{\phi\phi} \qquad = \quad -r\,\text{Sin}[\theta]^2$$

$$\Gamma^\theta{}_{tt} \qquad = \quad \frac{\text{Cos}[\theta]\,\text{Cos}[\phi]}{2r}$$

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

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

$$\Gamma^\phi{}_{tt} \qquad = \quad -\frac{\text{Csc}[\theta]\,\text{Sin}[\phi]}{2r}$$

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

Indeed, this is the exact same result we got when we transformed `SimpleMetricChristoffel` to spherical coordinates. We have learned an important lesson: since the Christoffel symbols do not transform like a tensor, we should always use the built-in module TCalcChristoffel to calculate them, which ensures that they transform properly. (Of course, this method is also much more convenient than writing the explicit definition...)

For future use, let us define the **Friedmann–Lemaitre–Robertson–Walker (FLRW) metric**, which

describes an expanding universe:

In[ ]:=
```
TSetReservedSymbols[{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}]]
```

OGRe:

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

This metric has the line element:

In[ ]:=
```
TLineElement["FLRW"]
```

Out[ ]=
$$-\mathrm{d}t^2 + a[t]^2 \left(\frac{\mathrm{d}r^2}{1 - k\,r^2} + r^2\,\mathrm{d}\theta^2 + r^2\,\mathrm{d}\phi^2\,\text{Sin}[\theta]^2 \right)$$

and the volume element squared:

In[ ]:=
```
TVolumeElementSquared["FLRW"]
```

Out[ ]=
$$\frac{r^4\,a[t]^6\,\text{Sin}[\theta]^2}{-1 + k\,r^2}$$

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 TCalcChristoffel:

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

FLRWChristoffel:

$$\Gamma^t_{\ rr} = \frac{a\,\partial_t a}{1 - k\,r^2}$$

$$\Gamma^t_{\ \theta\theta} = a\,r^2\,\partial_t a$$

$$\Gamma^t_{\ \phi\phi} = a\,r^2\,\text{Sin}[\theta]^2\,\partial_t a$$

$$\Gamma^r_{\ tr} = \Gamma^r_{\ rt} = \Gamma^\theta_{\ t\theta} = \Gamma^\theta_{\ \theta t} = \Gamma^\phi_{\ t\phi} = \Gamma^\phi_{\ \phi t} = \frac{\partial_t a}{a}$$

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

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

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

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

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

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

Notice how the derivatives of the function **a** are shown by `TList` in the notation $\partial_t a$ instead of **a'[t]**. This may not seem like much of an improvement in this case, but in the case of partial derivatives of functions of many arguments, it can greatly improve readability. Here is a simple example:

In[ ]:=
```
TShow@TNewTensor["PartialDerivatives", "FLRW", "Cartesian", {1}, {D[f[x, y, z], x],
    D[f[x, y, z], x, y], D[f[x, y, z], x, y, z], D[f[x, y, z], x, y, {z, 2}]}]
```

$$\text{PartialDerivatives:}\quad \square^\mu(t, x, y, z) = \begin{pmatrix} \partial_x f \\ \partial_{x,y} f \\ \partial_{x,y,z} f \\ \partial_{x,y,z^2} f \end{pmatrix}$$

Compare this with the raw components:

In[ ]:=
```
TGetComponents["PartialDerivatives"]
```

TGetComponents: Using the default index configuration {1} and the default coordinate system "Cartesian".

Out[ ]=
$$\left\{ f^{(1,0,0)}[x, y, z],\ f^{(1,1,0)}[x, y, z],\ f^{(1,1,1)}[x, y, z],\ f^{(1,1,2)}[x, y, z] \right\}$$

The Riemann tensor

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

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

Notice that even though it contains partial derivatives, it nonetheless transforms like a tensor under a change of coordinates, because the extra transformation terms exactly cancel each other. To calculate this tensor, we can simply write down the formula in **TCalc** with the correct indices 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}]
```

OGRe:

<div align="center">

SchwarzschildRiemann:

$R^{t}_{rtr} = -R^{t}_{rrt}$	$= \dfrac{2M}{r^2(-2M+r)}$
$R^{t}_{\theta t\theta} = R^{r}_{\theta r\theta} = -R^{t}_{\theta\theta t} = -R^{r}_{\theta\theta r}$	$= -\dfrac{M}{r}$
$R^{t}_{\phi t\phi} = R^{r}_{\phi r\phi} = -R^{t}_{\phi\phi t} = -R^{r}_{\phi\phi r}$	$= -\dfrac{M\sin[\theta]^2}{r}$
R^{r}_{ttr}	$= \dfrac{2M(-2M+r)}{r^4}$
R^{r}_{trt}	$= \dfrac{2M(2M-r)}{r^4}$
$R^{\theta}_{tt\theta} = R^{\phi}_{tt\phi}$	$= \dfrac{M(2M-r)}{r^4}$
$R^{\theta}_{t\theta t} = R^{\phi}_{t\phi t}$	$= \dfrac{M(-2M+r)}{r^4}$
$R^{\theta}_{rr\theta} = R^{\phi}_{rr\phi}$	$= \dfrac{M}{r^2(-2M+r)}$
$R^{\theta}_{r\theta r} = R^{\phi}_{r\phi r}$	$= \dfrac{M}{(2M-r)r^2}$
$R^{\theta}_{\phi\theta\phi} = -R^{\theta}_{\phi\phi\theta}$	$= \dfrac{2M\sin[\theta]^2}{r}$
$R^{\phi}_{\theta\theta\phi} = -R^{\phi}_{\theta\phi\theta}$	$= -\dfrac{2M}{r}$

</div>

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}]
```

OGRe:

$$\text{SchwarzschildRiemann:}$$

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

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

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

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

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

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

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

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

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

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

Don't worry - you don't need to write the explicit definition of the Riemann tensor every time we want to calculate it. Instead, OGRe offers the built-in module `TCalcRiemannTensor`:

In[ ]:=
```
? TCalcRiemannTensor
```

Out[ ]=

Symbol

TCalcRiemannTensor[**metricID**] calculates the Riemann tensor from the metric **metricID** and stores the result in a new 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, and will be used in the calculation. Otherwise, "**metricID**Christoffel" will be created using TCalcChristoffel[].

⌄

For example, for the FLRW metric we get:

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

<div style="border:1px solid #888; background:#e8e8e8; padding:10px;">

<center>FLRWRiemann:</center>

$$R^t{}_{rtr} = \frac{a\,\partial_{t^2}a}{1-k\,r^2}$$

$$R^t{}_{rrt} = \frac{a\,\partial_{t^2}a}{-1+k\,r^2}$$

$$R^t{}_{\theta t\theta} = -R^t{}_{\theta\theta t} = a\,r^2\,\partial_{t^2}a$$

$$R^t{}_{\phi t\phi} = -R^t{}_{\phi\phi t} = a\,r^2\,\text{Sin}[\theta]^2\,\partial_{t^2}a$$

$$R^r{}_{ttr} = R^\theta{}_{tt\theta} = R^\phi{}_{tt\phi} = -R^r{}_{trt} = -R^\theta{}_{t\theta t} = -R^\phi{}_{t\phi t} = \frac{\partial_{t^2}a}{a}$$

$$R^r{}_{\theta r\theta} = R^\phi{}_{\theta\phi\theta} = -R^r{}_{\theta\theta r} = -R^\phi{}_{\theta\theta\phi} = r^2\left(k+\partial_t a^2\right)$$

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

$$R^\theta{}_{rr\theta} = R^\phi{}_{rr\phi} = \frac{k+\partial_t a^2}{-1+k\,r^2}$$

$$R^\theta{}_{r\theta r} = R^\phi{}_{r\phi r} = \frac{k+\partial_t a^2}{1-k\,r^2}$$

</div>

Using `TCalcRiemannTensor` also has the advantage that it takes a metric as an input, and will automatically calculate the Christoffel symbols of the metric using `TCalcChristoffel` if they have not already been calculated. The same principle also applies to the other built-in modules for calculating curvature tensors, which we will present below; they always take a metric as input, and will calculate any intermediate tensors in their definitions automatically as needed.

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["KretschmannFromScratch",`
`    "SchwarzschildRiemann"["ρσμν"]."SchwarzschildRiemann"["ρσμν"], "K"]`

<div style="border:1px solid #888; background:#e8e8e8; padding:10px;">

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

</div>

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"]`

OGRe:

> ## Result:
> ## No non–zero elements.

We can also use the shorthand module `TCalcRicciTensor`:

In[ ]:= `? TCalcRicciTensor`

Out[ ]=

> Symbol
>
> TCalcRicciTensor[**metricID**] calculates the Ricci tensor from the metric **metricID** and stores the result in a new tensor object with ID "**metricID**RicciTensor". If a tensor with ID "**metricID**Riemann" exists, it will be assumed to be the Riemann tensor of the metric, and will be used in the calculation. Otherwise, "**metricID**Riemann" will be created using TCalcRiemannTensor[].
>
> ⌄

Here is the Ricci tensor for the FLRW metric:

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

OGRe:

$$
\begin{aligned}
\text{FLRWRicciTensor:} \\
R_{tt} &= -\frac{3\,\partial_{t^2} a}{a} \\
R_{rr} &= \frac{2\left(k+\partial_t a^2\right)+a\,\partial_{t^2} a}{1-k\,r^2} \\
R_{\theta\theta} &= r^2\left(2\left(k+\partial_t a^2\right)+a\,\partial_{t^2} a\right) \\
R_{\phi\phi} &= r^2\,\text{Sin}[\theta]^2\left(2\left(k+\partial_t a^2\right)+a\,\partial_{t^2} a\right)
\end{aligned}
$$

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"]`

OGRe:

$$\text{Result:} \quad R\,(t, r, \theta, \phi) = \frac{6\left(k + \partial_t a^2 + a\,\partial_{t^2} a\right)}{a^2}$$

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

In[ ]:=   `? TCalcRicciScalar`

Symbol

TCalcRicciScalar[**metricID** calculates the Ricci scalar from the metric **metricID** and stores the result in a new

Out[ ]=

tensor object with ID "**metricID**RicciScalar". If a tensor with ID "**metricID**RicciTensor" exists, it will be

assumed to be the Ricci tensor of the metric, and will be used in the calculation. Otherwise,

"**metricID**RicciTensor" will be created using TCalcRicciTensor[].

⌄

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

OGRe:

$$\text{FLRWRicciScalar:}$$

$$R \quad = \quad \frac{6\left(k + \partial_t a^2 + a\,\partial_{t^2} a\right)}{a^2}$$

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"]
```

OGRe:

<div align="center">

Result:

$G_{tt} \quad = \quad \dfrac{3\left(k+\partial_t a^2\right)}{a^2}$

$G_{rr} \quad = \quad \dfrac{k+\partial_t a^2 + 2\,a\,\partial_{t^2}a}{-1+k\,r^2}$

$G_{\theta\theta} \quad = \quad -r^2\left(k+\partial_t a^2 + 2\,a\,\partial_{t^2}a\right)$

$G_{\phi\phi} \quad = \quad -r^2\,\text{Sin}[\theta]^2\left(k+\partial_t a^2 + 2\,a\,\partial_{t^2}a\right)$

</div>

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

In[ ]:=
```
? TCalcEinsteinTensor
```

Out[ ]=

Symbol

TCalcEinsteinTensor[**metricID**] calculates the Einstein tensor from the metric **metricID** and stores the result in a new tensor object with ID "**metricID**Einstein". If a tensor with ID "**metricID**RicciTensor" exists, it will be assumed to be the Ricci tensor of the metric, and will be used in the calculation. Otherwise, "**metricID**RicciTensor" will be created using TCalcRicciTensor[].

⌄

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

OGRe:

<div align="center">

FLRWEinstein:

$G_{tt} \quad = \quad \dfrac{3\left(k+\partial_t a^2\right)}{a^2}$

$G_{rr} \quad = \quad \dfrac{k+\partial_t a^2 + 2\,a\,\partial_{t^2}a}{-1+k\,r^2}$

$G_{\theta\theta} \quad = \quad -r^2\left(k+\partial_t a^2 + 2\,a\,\partial_{t^2}a\right)$

$G_{\phi\phi} \quad = \quad -r^2\,\text{Sin}[\theta]^2\left(k+\partial_t a^2 + 2\,a\,\partial_{t^2}a\right)$

</div>

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 $T^{v_1 \cdots v_p}_{\sigma_1 \cdots \sigma_q}$, the covariant derivative $\nabla_\mu T^{v_1 \cdots v_p}_{\sigma_1 \cdots \sigma_q}$ is defined as follows:

- The first term will be $\partial_\mu T^{v_1 \cdots v_p}_{\sigma_1 \cdots \sigma_q}$.

- We **add** one term $\Gamma^{v_i}_{\mu\lambda} T^{v_1 \cdots \lambda \cdots v_p}_{\sigma_1 \cdots \sigma_q}$ for each upper index v_i.

- We **subtract** one term $\Gamma^\lambda_{\mu\sigma_i} T^{v_1 \cdots v_p}_{\sigma_1 \cdots \lambda \cdots \sigma_q}$ 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:

$$\nabla_\mu \mathbf{g}_{\alpha\beta} = \partial_\mu \mathbf{g}_{\alpha\beta} - \Gamma^\lambda_{\mu\alpha} \mathbf{g}_{\lambda\beta} - \Gamma^\lambda_{\mu\beta} \mathbf{g}_{\alpha\lambda} \cdot$$

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"]]
```

OGRe:

<div align="center">

Result:

No non-zero elements.

</div>

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
```

Symbol

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

Out[ ]=

tensor with ID "**metricID**Christoffel" exists, where **metricID** is the metric associated with the tensor the

derivative is acting upon, then it will be assumed to be the Levi–Civita connection of the metric, and will be

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

⌄

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

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

OGRe:

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"]]
```

OGRe:

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", "Spherical", {1}, {1, 0, 0, 0}, "u"]`

OGRe:

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

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

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

OGRe:

$$\text{PerfectFluidFLRW:} \quad T^{\mu\nu}(t, r, \theta, \phi) = \begin{pmatrix} \rho & 0 & 0 & 0 \\ 0 & \frac{p(1-kr^2)}{a^2} & 0 & 0 \\ 0 & 0 & \frac{p}{a^2 r^2} & 0 \\ 0 & 0 & 0 & \frac{p \, \text{Csc}[\theta]^2}{a^2 r^2} \end{pmatrix}$$

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

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

OGRe:

$$\text{FLRWConservation:}$$

$$\square^t \quad = \quad \frac{3(p+\rho)\,\partial_t a}{a} + \partial_t \rho$$

$$\square^r \quad = \quad \frac{(1-kr^2)\,\partial_r p}{a^2}$$

$$\square^\theta \quad = \quad \frac{\partial_\theta p}{a^2 r^2}$$

$$\square^\phi \quad = \quad \frac{\text{Csc}[\theta]^2\,\partial_\phi p}{a^2 r^2}$$

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.

Overwriting metrics

If overwriting tensors has been allowed using `TSetAllowOverwrite`, and we overwrite a metric tensor, then all of the curvature tensors calculated from the metric being overwritten will be automatically **deleted**, since the curvature tensors of the new metric will generally be different. For example, let us overwrite the FLRW metric with a similar metric where *k* is equal to 0:

In[ ]:=
```
currentOverwrite = TSetAllowOverwrite[];
TSetAllowOverwrite[True];
TShow@TNewMetric["FLRW", "Spherical",
    DiagonalMatrix[{-1, a[t]², a[t]² r², a[t]² r² Sin[θ]²}]]
```

TSetAllowOverwrite: Overwriting tensors turned on.

TMessage: Overwriting the tensor "FLRW".

TNewMetric: All curvature tensors previously calculated from the metric being overwritten will be deleted.

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

If we now try to access, for example, **"FLRWChristoffel"**, we will discover that it has been deleted:

In[ ]:=
```
TShow["FLRWChristoffel"]
```

TMessage: The tensor "FLRWChristoffel" does not exist.

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

We must recalculate this curvature tensor, and all others, since they will be different from the ones we calculated from the previous metric.

Finally, let us reset `TSetAllowOverwrite` back to its previous setting:

In[ ]:=
```
TSetAllowOverwrite[currentOverwrite]
```

TSetAllowOverwrite: Overwriting tensors turned on.

Curves and geodesics

The Curve Lagrangian

Consider a **curve**, which is a function $x^\mu(\lambda)$ on the manifold where λ is called the **curve parameter**. The **curve Lagrangian** of a metric is defined as the norm-squared of the tangent to the curve:

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

where $\dot{x}^\mu$ is the first derivative of x^μ with respect to the curve parameter (in Newton dot notation). We can calculate it using the module `TCalcLagrangian`:

In[ ]:=   `? TCalcLagrangian`

Symbol

TCalcLagrangian[**metricID**, **coordinatesID**] calculates the curve Lagrangian of the metric **metricID**, defined as the norm−squared of the tangent to the curve, and stores the result in a new tensor object with ID "**metricID**Lagrangian".

Taking the square root of (the absolute value of) the Lagrangian yields the integrand of the curve length

Out[ ]=   functional. Varying the Lagrangian using the Euler−Lagrange equations yields the geodesic equations.

The result will be given in terms of the coordinate symbols as functions of the curve parameter and their derivatives with respect to the curve parameter. The curve parameter can be selected using TSetCurveParameter[].

If **coordinatesID** is not specified, the default coordinate system of the metric will be used.

⌄

For example:

In[ ]:=   `TList@TCalcLagrangian["Minkowski"]`

MinkowskiLagrangian:

OGRe:

$$L \;=\; -\dot{t}^2 + \dot{x}^2 + \dot{y}^2 + \dot{z}^2$$

In[ ]:=   `TList@TCalcLagrangian["Schwarzschild"]`

SchwarzschildLagrangian:

OGRe:

$$L \;=\; \frac{\dot{r}^2}{1-\frac{2M}{r}} + \left(-1+\frac{2M}{r}\right)\dot{t}^2 + r^2\left(\dot{\theta}^2 + \dot{\phi}^2\, \mathrm{Sin}[\theta]^2\right)$$

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

OGRe:
FLRWLagrangian:
$$L = -\dot{t}^2 + a^2\left(\dot{r}^2 + r^2\left(\dot{\theta}^2 + \dot{\phi}^2\,\text{Sin}[\theta]^2\right)\right)$$

In[ ]:=
```
TList@TCalcLagrangian["Alcubierre"]
```

OGRe:
AlcubierreLagrangian:
$$L = \left(-1 + f^2\,v^2\right)\dot{t}^2 + \dot{x}^2 + \dot{y}^2 - 2\,f\,v\,\dot{t}\,\dot{z} + \dot{z}^2$$

Notice how **TList** (and **TShow**) use Newton dot notation for the derivatives of the coordinate functions, for improved readability. To get the full expressions with the explicit derivatives, we can use **TGetComponents**. For example:

In[ ]:=
```
TGetComponents["MinkowskiLagrangian"]
```

TGetComponents: Using the default index configuration {} and the default coordinate system "Cartesian".

Out[ ]=
$$\left\{-t'[\lambda]^2 + x'[\lambda]^2 + y'[\lambda]^2 + z'[\lambda]^2\right\}$$

Geodesic equations from the Lagrangian

By applying the Euler-Lagrange equations to the curve Lagrangian:
$$\frac{d}{d\lambda}\left(\frac{\partial L}{\partial \dot{x}^\mu}\right) - \frac{\partial L}{\partial x^\mu} = 0,$$

we can obtain the geodesic equations for our spacetime. This is done using the module **TCalcGeodesicFromLagrangian**:

In[ ]:= `? TCalcGeodesicFromLagrangian`

Out[ ]=

Symbol

TCalcGeodesicFromLagrangian[**metricID**, **coordinatesID**] calculates the geodesic equations obtained for each of the coordinates in **coordinatesID** using the curve Lagrangian of the metric **metricID** and stores the result in a new rank–1 tensor object with ID "**metricID**GeodesicFromLagrangian". Equating the components to zero will yield the full system of geodesic equations.

Derivatives with respect to the curve parameter in the Euler–Lagrange equation will be left unevaluated using Inactive[], which can sometimes help solve the geodesic equations by inspection. Use Activate[] to evaluate the derivatives.

The result will be given in terms of the coordinate symbols as functions of the curve parameter and their derivatives with respect to the curve parameter. The curve parameter can be selected using TSetCurveParameter[].

If **coordinatesID** is not specified, the default coordinate system of the metric will be used.

⌄

In the Minkowski metric, the geodesic equations are:

In[ ]:= `TList@TCalcGeodesicFromLagrangian["Minkowski"]`

OGRe:

$$\text{MinkowskiGeodesicFromLagrangian:}$$
$$0^t = -\partial_\lambda\left(-\dot{t}\right)$$
$$0^x = -\partial_\lambda\dot{x}$$
$$0^y = -\partial_\lambda\dot{y}$$
$$0^z = -\partial_\lambda\dot{z}$$

Note that this module only calculates the left-hand side of the Euler-Lagrange equations; if we equate the result to zero, we will get the actual geodesics equations. This is hinted at visually by setting the resulting tensor's symbol to 0, so that you actually see the equations when using `TList`. It is trivial to see that the solution to these equations is simply a curve with a constant velocity; in a flat Minkowski spacetime, particles experience no gravitational force, and thus no acceleration (unless some other force acts on them, of course).

The derivatives with respect to the curve parameter λ are kept unevaluated, using the Mathematica function `Inactive`. This simplifies the equations, and can sometimes help solve them by inspection. If we want to activate the derivatives, we simply need to use `Activate`. Recall that `TList` and `TShow` can apply a function to the tensor's components before displaying them, so we just need to pass `Activate` as the last argument:

In[ ]:= `TList["MinkowskiGeodesicFromLagrangian", Activate]`

$$\text{MinkowskiGeodesicFromLagrangian:}$$
$$0^t \;=\; \ddot{t}$$
$$0^x \;=\; -\ddot{x}$$
$$0^y \;=\; -\ddot{y}$$
$$0^z \;=\; -\ddot{z}$$

Now the derivatives have been activated.

As with the Lagrangian itself, the geodesic equations are displayed in compact notation when using **TList**. If we want the full expressions with the explicit derivatives, for example in order to pass them to **DSolve** and actually solve the equations, we can use **TGetComponents**:

In[ ]:= `TGetComponents["MinkowskiGeodesicFromLagrangian"]`

TGetComponents: Using the default index configuration {1} and the default coordinate system "Cartesian".

Out[ ]= $\{-\partial_\lambda(-t'[\lambda]), -\partial_\lambda x'[\lambda], -\partial_\lambda y'[\lambda], -\partial_\lambda z'[\lambda]\}$

Again, if we wish the activate the derivatives, we need to use **Activate**:

In[ ]:= `TGetComponents["MinkowskiGeodesicFromLagrangian", Activate]`

TGetComponents: Using the default index configuration {1} and the default coordinate system "Cartesian".

Out[ ]= $\{t''[\lambda], -x''[\lambda], -y''[\lambda], -z''[\lambda]\}$

We can similarly find the geodesic equations of other metrics:

In[ ]:= `TList@TCalcGeodesicFromLagrangian["Schwarzschild"]`

$$\text{SchwarzschildGeodesicFromLagrangian:}$$
$$0^t \;=\; -\partial_\lambda\!\left(\left(-1+\tfrac{2M}{r}\right)\dot{t}\right)$$
$$0^r \;=\; -\frac{M\dot{r}^2}{(-2M+r)^2} - \frac{M\dot{t}^2}{r^2} + r\left(\dot{\theta}^2 + \dot{\phi}^2\,\text{Sin}[\theta]^2\right) - \partial_\lambda\frac{\dot{r}}{1-\frac{2M}{r}}$$
$$0^\theta \;=\; r^2\,\text{Cos}[\theta]\,\dot{\phi}^2\,\text{Sin}[\theta] - \partial_\lambda\!\left(r^2\,\dot{\theta}\right)$$
$$0^\phi \;=\; -\partial_\lambda\!\left(r^2\,\dot{\phi}\,\text{Sin}[\theta]^2\right)$$

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

OGRe:

FLRWGeodesicFromLagrangian:

$$0^t \;=\; a\left(\dot{r}^2 + r^2\left(\dot{\theta}^2 + \dot{\phi}^2 \operatorname{Sin}[\theta]^2\right)\right)\partial_t a - \partial_\lambda(-\dot{t})$$

$$0^r \;=\; a^2\, r\left(\dot{\theta}^2 + \dot{\phi}^2 \operatorname{Sin}[\theta]^2\right) - \partial_\lambda(a^2\, \dot{r})$$

$$0^\theta \;=\; a^2\, r^2 \operatorname{Cos}[\theta]\, \dot{\phi}^2 \operatorname{Sin}[\theta] - \partial_\lambda(a^2\, r^2\, \dot{\theta})$$

$$0^\phi \;=\; -\partial_\lambda(a^2\, r^2\, \dot{\phi} \operatorname{Sin}[\theta]^2)$$

In[ ]:= `TList@TCalcGeodesicFromLagrangian["Alcubierre"]`

OGRe:

AlcubierreGeodesicFromLagrangian:

$$0^t \;=\; \dot{t}\left(f\, v\, \dot{t} - \dot{z}\right)\left(v\, \partial_t f + f\, \partial_t v\right) - \partial_\lambda\left(\left(-1 + f^2\, v^2\right)\dot{t} - f\, v\, \dot{z}\right)$$

$$0^x \;=\; v\, \dot{t}\left(f\, v\, \dot{t} - \dot{z}\right)\partial_x f - \partial_\lambda \dot{x}$$

$$0^y \;=\; v\, \dot{t}\left(f\, v\, \dot{t} - \dot{z}\right)\partial_y f - \partial_\lambda \dot{y}$$

$$0^z \;=\; v\, \dot{t}\left(f\, v\, \dot{t} - \dot{z}\right)\partial_z f - \partial_\lambda\left(-f\, v\, \dot{t} + \dot{z}\right)$$

The last example, the geodesics equations of the Alcubierre metric, is a good example of how we can solve the geodesic equations by inspection. Indeed, it is easy to see that

$$\dot{x}^\mu \;=\; (\texttt{1, 0, 0, vf})$$

is a solution to this system of equations, since then we have $\dot{x} = \dot{y} = 0$ and $\left(f\, v\, \dot{t} - \dot{z}\right) = 0$, and both terms in each equation vanish (the last term in the first equation will reduce to $\partial_\lambda(-1)$, which is of course zero). We can check this solution by replacing the coordinate functions with their solutions; since we will be left with $\partial_\lambda(-1)$ in the first equation, we must also activate the derivative:

In[ ]:= `TList["AlcubierreGeodesicFromLagrangian",`
`  (Activate@ReplaceAll[#, {t'[λ] → 1, x'[λ] → 0,`
`      y'[λ] → 0, z'[λ] → v[t[λ]] × f[t[λ], x[λ], y[λ], z[λ]]}]) &]`

OGRe:

AlcubierreGeodesicFromLagrangian:

No non−zero elements.

Notice how we had to write the coordinates explicitly as **functions of the curve parameter**, even when they are arguments of a function; for example, `v[t]` became `v[t[λ]]`. This solution indicates that we are traveling with velocity *v* in the *z* direction; the warp bubble (inside which, as you recall, *f* = 1) moves whatever is inside it, such as a spaceship, through space at the velocity *v*, but there is no limit on *v* - it can even be faster than light!

Geodesic equations from the Christoffel symbols

Another way of obtaining the geodesic equations is using the covariant derivative, and thus the Christoffel symbols:

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

In OGRe, we can calculate the left-hand side of this equation using `TCalcGeodesicFromChristoffel`:

In[ ]:= **? TCalcGeodesicFromChristoffel**

Symbol

Out[ ]= TCalcGeodesicFromChristoffel[**metricID**, **coordinatesID**] calculates the geodesic equations obtained for each of the coordinates in **coordinatesID** using the Christoffel symbols of the metric **metricID** and stores the result in a new rank–1 tensor object with ID "**metricID**GeodesicFromChristoffel". Equating the components to zero will yield the full system of geodesic equations.

The result will be given in terms of the coordinate symbols as functions of the curve parameter and their derivatives with respect to the curve parameter. The curve parameter can be selected using TSetCurveParameter[].

If **coordinatesID** is not specified, the default coordinate system of the metric will be used.

For example:

In[ ]:= **TList@TCalcGeodesicFromChristoffel["Minkowski"]**

OGRe:

$$\text{MinkowskiGeodesicFromChristoffel:}$$

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

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

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

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

In[ ]:= `TList@TCalcGeodesicFromChristoffel["Schwarzschild"]`

OGRe:

$$\text{SchwarzschildGeodesicFromChristoffel:}$$

$$0^t \;=\; \frac{2\,M\,\dot{r}\,\dot{t}}{r\,(-2\,M+r)} + \ddot{t}$$

$$0^r \;=\; \frac{M\,\dot{r}^2}{2\,M\,r - r^2} + \frac{M\,(-2\,M+r)\,\dot{t}^2}{r^3} + \ddot{r} + (2\,M - r)\left(\dot{\theta}^2 + \dot{\phi}^2\,\text{Sin}[\theta]^2\right)$$

$$0^\theta \;=\; \frac{2\,\dot{r}\,\dot{\theta}}{r} + \ddot{\theta} - \text{Cos}[\theta]\,\dot{\phi}^2\,\text{Sin}[\theta]$$

$$0^\phi \;=\; \frac{2\left(\dot{r} + r\,\text{Cot}[\theta]\,\dot{\theta}\right)\dot{\phi}}{r} + \ddot{\phi}$$

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

OGRe:

$$\text{FLRWGeodesicFromChristoffel:}$$

$$0^t \;=\; \ddot{t} + a\left(\dot{r}^2 + r^2\left(\dot{\theta}^2 + \dot{\phi}^2\,\text{Sin}[\theta]^2\right)\right)\partial_t a$$

$$0^r \;=\; \ddot{r} - r\left(\dot{\theta}^2 + \dot{\phi}^2\,\text{Sin}[\theta]^2\right) + \frac{2\,\dot{r}\,\dot{t}\,\partial_t a}{a}$$

$$0^\theta \;=\; \ddot{\theta} - \text{Cos}[\theta]\,\dot{\phi}^2\,\text{Sin}[\theta] + 2\,\dot{\theta}\left(\frac{\dot{r}}{r} + \frac{\dot{t}\,\partial_t a}{a}\right)$$

$$0^\phi \;=\; \ddot{\phi} + 2\,\dot{\phi}\left(\frac{\dot{r}}{r} + \text{Cot}[\theta]\,\dot{\theta} + \frac{\dot{t}\,\partial_t a}{a}\right)$$

In[ ]:= `TList@TCalcGeodesicFromChristoffel["Alcubierre"]`

OGRe:

$$\text{AlcubierreGeodesicFromChristoffel:}$$

$$0^t \;=\; \ddot{t} + v\left(f\,v\,\dot{t} - \dot{z}\right)\left(-\dot{x}\,\partial_x f - \dot{y}\,\partial_y f + \left(f\,v\,\dot{t} - \dot{z}\right)\partial_z f\right)$$

$$0^x \;=\; \ddot{x} + v\,\dot{t}\left(-f\,v\,\dot{t} + \dot{z}\right)\partial_x f$$

$$0^y \;=\; \ddot{y} + v\,\dot{t}\left(-f\,v\,\dot{t} + \dot{z}\right)\partial_y f$$

$$0^z \;=\; \ddot{z} - v\,\dot{t}\left(\dot{t}\,\partial_t f + \dot{x}\,\partial_x f + \dot{y}\,\partial_y f\right) +$$
$$f^3\,v^4\,\dot{t}^2\,\partial_z f - f^2\,v^3\,\dot{t}\left(\dot{x}\,\partial_x f + \dot{y}\,\partial_y f + 2\,\dot{z}\,\partial_z f\right) +$$
$$f\left(v^2\,\dot{z}\left(\dot{x}\,\partial_x f + \dot{y}\,\partial_y f + \dot{z}\,\partial_z f\right) - \dot{t}^2\left(v^2\,\partial_z f + \partial_t v\right)\right)$$

Often, you will find that the Lagrangian method produces simpler equations, which can even be solved by inspection, as we did for the Alcubierre metric. This is due to the possibility of leaving the λ derivative unevaluated. However, in other cases, the Christoffel method might produce simpler equations. The best thing to do is to try both methods and see which one produces simpler or nicer results for the

specific metric in question. Note that the system of equations obtained using `TCalcGeodesicFromLagrangian` will often be different from the one obtained using `TCalcGeodesicFromChristoffel`, but both systems will always have the same solutions.

Changing the curve parameter

By default, the curve parameter is λ. However, sometimes we want to use another parameter - for example τ, the proper time. To change the parameter, we use `TSetCurveParameter`:

In[ ]:=
```
? TSetCurveParameter
```

Out[ ]=

Symbol

TSetCurveParameter[] shows the curve parameter used for calculating Lagrangians and geodesics.

TSetCurveParameter[**parameter**] changes the curve parameter. The new parameter will be cleared and protected, and the old parameter will be unprotected. Any tensors currently using the old parameter will be modified to use the new parameter. **parameter** can be given either as a symbol name or a string representing a symbol name.

TSetCurveParameter[Automatic] resets the curve parameter to the default: λ.

∨

Let us change it to τ:

In[ ]:=
```
TSetCurveParameter[τ];
```

This changes the curve parameter **retroactively**, so that if we get the components of any Lagrangian or geodesic equation we previously calculated, the parameter will now be τ instead of λ:

In[ ]:=
```
TGetComponents["MinkowskiLagrangian"]
```

TGetComponents: Using the default index configuration {} and the default coordinate system "Cartesian".

Out[ ]=
$$\left\{ -t'[\tau]^2 + x'[\tau]^2 + y'[\tau]^2 + z'[\tau]^2 \right\}$$

When exporting tensors using `TExportAll`, the choice of curve parameter is exported as well, so that when you import the tensors later using `TImportAll`, OGRe will automatically know which curve parameter was used when calculating the imported tensors.

Geodesics equations in terms of the time coordinate

If the metric is a spacetime metric, it is often convenient to obtain the geodesic equations in terms of the time parameter, instead of an affine curve parameter. It can be shown that the geodesic equations in terms of the time coordinate are given by

$$\frac{d^2 x^\sigma}{dt^2} + \left(\Gamma^\sigma_{\mu\nu} - \Gamma^0_{\mu\nu} \frac{dx^\sigma}{dt} \right) \frac{dx^\mu}{dt} \frac{dx^\nu}{dt} = 0,$$

where we are assuming the time coordinate is t and it is the first coordinate. These equations can be obtained using `TCalcGeodesicWithTimeParameter`:

In[ ]:=
```
? TCalcGeodesicWithTimeParameter
```

Out[ ]=

> Symbol
>
> TCalcGeodesicWithTimeParameter[**metricID**, **coordinatesID**] calculates the geodesic equations obtained for each of the coordinates in **coordinatesID** using the Christoffel symbols of the metric **metricID** and stores the result in a new rank–1 tensor object with ID "**metricID**GeodesicFromChristoffel". Equating the components to zero will yield the full system of geodesic equations.
>
> The result will be given in terms of the spatial coordinate symbols as functions of the time coordinate and their derivatives with respect to time. The first coordinate of **coordinatesID** will be assumed to be the time coordinate, even if its symbol is not t.
>
> If **coordinatesID** is not specified, the default coordinate system of the metric will be used.
>
> ⌄

Note that `TCalcGeodesicWithTimeParameter` assumes time is the first coordinate, but the coordinate does not need to have the symbol t. As an example, the equations for the FLRW metric in terms of a curve parameter are

In[ ]:=
```
TList@TCalcGeodesicFromChristoffel["FLRW", "Cartesian"]
```

TMessage: Overwriting the tensor "FLRWGeodesicFromChristoffel".

OGRe:

$$\text{FLRWGeodesicFromChristoffel:}$$

$$0^t = \ddot{t} + a \left(\dot{x}^2 + \dot{y}^2 + \dot{z}^2 \right) \partial_t a$$

$$0^x = \ddot{x} + \frac{2\,\dot{t}\,\dot{x}\,\partial_t a}{a}$$

$$0^y = \ddot{y} + \frac{2\,\dot{t}\,\dot{y}\,\partial_t a}{a}$$

$$0^z = \ddot{z} + \frac{2\,\dot{t}\,\dot{z}\,\partial_t a}{a}$$

but in terms of t, we only need 3 equations:

In[ ]:=
```
TList@TCalcGeodesicWithTimeParameter["FLRW", "Cartesian"]
```

OGRe:

$$\text{FLRWGeodesicWithTimeParameter:}$$

$$0^x = \ddot{x} - \frac{\dot{x}\left(-2+a^2\left(\dot{x}^2+\dot{y}^2+\dot{z}^2\right)\right)\partial_t a}{a}$$

$$0^y = \ddot{y} - \frac{\dot{y}\left(-2+a^2\left(\dot{x}^2+\dot{y}^2+\dot{z}^2\right)\right)\partial_t a}{a}$$

$$0^z = \ddot{z} - \frac{\dot{z}\left(-2+a^2\left(\dot{x}^2+\dot{y}^2+\dot{z}^2\right)\right)\partial_t a}{a}$$

These equations are easier to solve. For simplicity, assume that we are only moving along the x coordinate. Then we only have one equation to solve:

In[ ]:=
```
FLRWEq =
  TGetComponents["FLRWGeodesicWithTimeParameter", ReplaceAll[y'[t]² + z'[t]² → 0]]〚2〛
```

TGetComponents: Using the default index configuration {1} and the default coordinate system "Cartesian".

Out[ ]=

$$\frac{a'[t]\,x'[t]\,\left(2 - a[t]^2\,x'[t]^2\right)}{a[t]} + x''[t]$$

The solution can be obtained in terms of an integral over a(t):

In[ ]:=
```
FLRWSol = DSolve[FLRWEq == 0, x, t]
```

Out[ ]=

$$\left\{\left\{x \to \text{Function}\left[\{t\}, \mathbb{c}_2 + \int_1^t -\frac{1}{\sqrt{a[K[1]]^2 + a[K[1]]^4\,\mathbb{c}_1}}\,\mathrm{d}K[1]\right]\right\},\right.$$

$$\left.\left\{x \to \text{Function}\left[\{t\}, \mathbb{c}_2 + \int_1^t \frac{1}{\sqrt{a[K[2]]^2 + a[K[2]]^4\,\mathbb{c}_1}}\,\mathrm{d}K[2]\right]\right\}\right\}$$

From this we can get the coordinate velocity $\dot{x}$ along x:

In[ ]:=
```
x'[t] == FullSimplify[a[t] × D[x[t] /. FLRWSol〚2〛], t], a[t] > 0] / a[t]
```

Out[ ]=

$$x'[t] == \frac{1}{a[t]\,\sqrt{1 + a[t]^2\,\mathbb{c}_1}}$$

More about tensor calculations

Calculating the norm squared

The norm squared of a tensor of any rank can be calculated with **TCalcNormSquared**:

In[ ]:=   **? TCalcNormSquared**

Out[ ]=

> Symbol
>
> TCalcNormSquared[**tensorID** calculates the norm–squared of the tensor **tensorID** with respect to its metric, that is, the tensor contracted with itself in all indices. For example, for a vector v^a the norm–squared will be v^a v_a and for a rank–2 tensor T^ab the result will be T^ab T_ab. The result is stored in a new rank–0 tensor object with ID "**tensorID**NormSquared".
>
> ⌄

For any tensor, the norm squared is defined as the tensor contracted with itself in all indices:

$$|v|^2 \equiv v^\mu v_\mu,$$
$$|T|^2 \equiv T^{\mu\nu} T_{\mu\nu},$$

and so on. The result is always a scalar. For example, the norm of any 4-velocity is -1:

In[ ]:=   **TList@TCalcNormSquared["4-Velocity"]**

OGRe:

$$4\text{–VelocityNormSquared:}$$
$$\square = -1$$

The norm of any metric is the number of dimensions in that metric:

In[ ]:=   **TList@TCalcNormSquared["Minkowski"]**

OGRe:

$$\text{MinkowskiNormSquared:}$$
$$\square = 4$$

The Kretschmann scalar is the norm squared of the Riemann tensor:

In[ ]:=   **TList@TCalcNormSquared["SchwarzschildRiemann"]**

OGRe:

$$\text{SchwarzschildRiemannNormSquared:}$$
$$\square = \frac{48 M^2}{r^6}$$

Parallelization

Improving performance with parallelization

The calculations we have demonstrated so far in this documentation have been quite simple, and should not take more than a second to perform on a decent computer. However, when doing research, calculations can be much more involved, and thus also take more time to complete. Typically, the most time-consuming part of any tensor calculation is not the tensor operations themselves, but rather the **simplification** of the final result using `FullSimplify`.

If simplification is taking more than a few seconds, it is highly recommended to turn on the **parallelization** feature, which simplifies the components of the tensors in parallel instead of one after the other. This can provide a significant performance boost, proportional to the number of parallel kernels that can be launched. Note that this number is determined by your Mathematica license, and it may be less than the number of cores in your CPU.

Since we would like to measure execution time in this section, let us use the command `ClearSystemCache` to clear Mathematica's cache - since otherwise it will remember the results of previous simplifications, which may artificially speed up the calculation:

In[ ]:=
```
ClearSystemCache[]
```

To demonstrate the benefits of parallelization, let us consider the following somewhat complicated non-diagonal metric, which depends on an abstract function `f`:

In[ ]:=
```
TShow[TNewMetric["ParallelizationTest",
   "Cartesian", Table[f[a b t² + x³], {a, 1, 4}, {b, 1, 4}]]]
```

OGRe:

$$\text{ParallelizationTest: } g_{\mu\nu}(t, x, y, z$$

$$\begin{pmatrix} f[t^2 + x^3] & f[2\,t^2 + x^3] & f[3\,t^2 + x^3] & f[\\ f[2\,t^2 + x^3] & f[4\,t^2 + x^3] & f[6\,t^2 + x^3] & f[\\ f[3\,t^2 + x^3] & f[6\,t^2 + x^3] & f[9\,t^2 + x^3] & f[1 \\ f[4\,t^2 + x^3] & f[8\,t^2 + x^3] & f[12\,t^2 + x^3] & f[1 \end{pmatrix}$$

We first calculate the Christoffel symbols for this metric **without** parallelization, and use `AbsoluteTiming` to measure how long it takes:

In[ ]:=
```
notParallelChristoffel = AbsoluteTiming[TCalcChristoffel["ParallelizationTest"]]
```

Out[ ]=
```
{32.2502, ParallelizationTestChristoffel}
```

The first item on the list is the number of seconds this calculation took on my computer. On your computer this duration may be shorter or longer, but the calculation will invariably take a considerable amount of time on any personal computer. The vast majority of that time is spent not on calculating the tensor, but on simplifying the result.

To turn on parallelization, we use the module `TSetParallelization`:

In[ ]:=
> **? TSetParallelization**

Out[ ]=
> Symbol
>
> TSetParallelization[**True**] enables the parallelization of tensor simplifications, and TSetParallelization[**False**] disables it. The default value is **False**. TSetParallelization[] returns the current value. If simplifications take less than a few seconds, then you should leave parallelization off, as it has a small overhead and may actually impede performance. However, if simplifications are taking more than a few seconds, then it is highly recommended to enable parallelization for a significant performance boost.
>
> ⌄

Let us turn it on now:

In[ ]:=
> **TSetParallelization[True]**

OGRe:
> Parallelization enabled.

OGRe:
> 8 parallel kernels launched. CPU has 12 cores.

When parallelization is first turned on, all available parallel kernels are automatically launched. As you can see, my system has a 12-core CPU, but only 8 parallel kernels can be launched, since that is what my Mathematica license allows.

We will now repeat the calculation of the Christoffel symbols, in order to see how its performance improves with parallelization. However, before we can do that, we must again use the command **ClearSystemCache** to clear Mathematica's cache:

In[ ]:=
> **ClearSystemCache[]**

(If you don't trust that **ClearSystemCache** is enough to make a reliable benchmark, you can exit the kernel using **Quit**, and then reload the package and redefine the Cartesian coordinates and the test metric.)

We will also delete the tensor that we calculated previously, so that we can calculate it again (if overwriting is not turned on):

In[ ]:=
> **TDelete["ParallelizationTestChristoffel"];**

Now we can accurately measure the execution time for TCalcChristoffel with parallelization:

In[ ]:=
> **parallelChristoffel = AbsoluteTiming[TCalcChristoffel["ParallelizationTest"]]**

Out[ ]=
> {8.91397, ParallelizationTestChristoffel}

We see that the calculation now took less time, with an improvement by a factor of:

In[ ]:=   `notParallelChristoffel[[1]] / parallelChristoffel[[1]]`

Out[ ]=   `3.61794`

With longer calculations, the improvement will be even more significant. Increasing the number of kernels (if it was allowed by my license) would provide an additional speedup.

As a rule of thumb, if simplifications are taking less than a few seconds, then you should leave parallelization off, as it has a small overhead and may actually impede performance in that case. However, if simplifications are taking more than a few seconds, then it is highly recommended to enable parallelization for a significant performance boost.

Making use of OGRe's parallelized simplifications for other Mathematica expressions

When `TSimplify` is applied to any expression which is not a tensor object, the object is simplified based on the user-defined simplification assumptions set using `TSetAssumptions`:

In[ ]:=   `TSetAssumptions[]`

Out[ ]= $\langle|\,\text{AssumeReal} \to \text{True, User} \to \{r \geq 0\}\,|\rangle$

In[ ]:=   `TSimplify`$\left[\sqrt{r^2}\right]$

Out[ ]=   `r`

If the expression is a `List`, the components will be simplified in parallel. The user can thus make use of OGRe's optimized simplification process to simplify any Mathematica expression. For example, the following simplification will be slow and will not use our simplification assumptions:

In[ ]:=   `testExpression = Table`$\left[f\left[a\,b\,\sqrt{r^{a+b}}\right]\right.$`, {a, 1, 5}, {b, 1, 5}`$\Big]$`;`
            `ClearSystemCache[]`
            `notParallelSimplify = AbsoluteTiming[FullSimplify[testExpression.testExpression]]`

Out[ ]= $\left\{1.41976,\ \left\{\left\{f\left[\sqrt{r^2}\right]^2 + f\left[2\sqrt{r^3}\right]^2 + f\left[3\sqrt{r^4}\right]^2 + f\left[4\sqrt{r^5}\right]^2 + f\left[5\sqrt{r^6}\right]^2,\right.\right.\right.$

$f\left[2\sqrt{r^3}\right] \times \left(f\left[\sqrt{r^2}\right] + f\left[4\sqrt{r^4}\right]\right) + f\left[3\sqrt{r^4}\right] \times f\left[6\sqrt{r^5}\right] +$

$f\left[4\sqrt{r^5}\right] \times f\left[8\sqrt{r^6}\right] + f\left[5\sqrt{r^6}\right] \times f\left[10\sqrt{r^7}\right],\ f\left[2\sqrt{r^3}\right] \times f\left[6\sqrt{r^5}\right] +$

$f\left[3\sqrt{r^4}\right] \times \left(f\left[\sqrt{r^2}\right] + f\left[9\sqrt{r^6}\right]\right) + f\left[4\sqrt{r^5}\right] \times f\left[12\sqrt{r^7}\right] + f\left[5\sqrt{r^6}\right] \times f\left[15\sqrt{r^8}\right],$

$f\left[2\sqrt{r^3}\right] \times f\left[8\sqrt{r^6}\right] + f\left[3\sqrt{r^4}\right] \times f\left[12\sqrt{r^7}\right] + f\left[4\sqrt{r^5}\right] \times \left(f\left[\sqrt{r^2}\right] + f\left[16\sqrt{r^8}\right]\right) +$

$f\left[5\sqrt{r^6}\right] \times f\left[20\sqrt{r^9}\right],\ f\left[2\sqrt{r^3}\right] \times f\left[10\sqrt{r^7}\right] + f\left[3\sqrt{r^4}\right] \times f\left[15\sqrt{r^8}\right] +$

$f\left[4\sqrt{r^5}\right] \times f\left[20\sqrt{r^9}\right] + f\left[5\sqrt{r^6}\right] \times \left(f\left[\sqrt{r^2}\right] + f\left[25\sqrt{r^{10}}\right]\right)\right\},$

$$\Big\{ f\left[2\sqrt{r^3}\right] \times \left(f\left[\sqrt{r^2}\right] + f\left[4\sqrt{r^4}\right]\right) + f\left[3\sqrt{r^4}\right] \times f\left[6\sqrt{r^5}\right] + f\left[4\sqrt{r^5}\right] \times f\left[8\sqrt{r^6}\right] +$$
$$f\left[5\sqrt{r^6}\right] \times f\left[10\sqrt{r^7}\right], \ f\left[2\sqrt{r^3}\right]^2 + f\left[4\sqrt{r^4}\right]^2 + f\left[6\sqrt{r^5}\right]^2 + f\left[8\sqrt{r^6}\right]^2 + f\left[10\sqrt{r^7}\right]^2,$$
$$f\left[2\sqrt{r^3}\right] \times f\left[3\sqrt{r^4}\right] + f\left[6\sqrt{r^5}\right] \times \left(f\left[4\sqrt{r^4}\right] + f\left[9\sqrt{r^6}\right]\right) + f\left[8\sqrt{r^6}\right] \times f\left[12\sqrt{r^7}\right] +$$
$$f\left[10\sqrt{r^7}\right] \times f\left[15\sqrt{r^8}\right], \ f\left[2\sqrt{r^3}\right] \times f\left[4\sqrt{r^5}\right] + f\left[6\sqrt{r^5}\right] \times f\left[12\sqrt{r^7}\right] +$$
$$f\left[8\sqrt{r^6}\right] \times \left(f\left[4\sqrt{r^4}\right] + f\left[16\sqrt{r^8}\right]\right) + f\left[10\sqrt{r^7}\right] \times f\left[20\sqrt{r^9}\right], \ f\left[2\sqrt{r^3}\right] \times f\left[5\sqrt{r^6}\right] +$$
$$f\left[6\sqrt{r^5}\right] \times f\left[15\sqrt{r^8}\right] + f\left[8\sqrt{r^6}\right] \times f\left[20\sqrt{r^9}\right] + f\left[10\sqrt{r^7}\right] \times \left(f\left[4\sqrt{r^4}\right] + f\left[25\sqrt{r^{10}}\right]\right)\Big\},$$
$$\Big\{ f\left[2\sqrt{r^3}\right] \times f\left[6\sqrt{r^5}\right] + f\left[3\sqrt{r^4}\right] \times \left(f\left[\sqrt{r^2}\right] + f\left[9\sqrt{r^6}\right]\right) + f\left[4\sqrt{r^5}\right] \times f\left[12\sqrt{r^7}\right] +$$
$$f\left[5\sqrt{r^6}\right] \times f\left[15\sqrt{r^8}\right], \ f\left[2\sqrt{r^3}\right] \times f\left[3\sqrt{r^4}\right] + f\left[6\sqrt{r^5}\right] \times \left(f\left[4\sqrt{r^4}\right] + f\left[9\sqrt{r^6}\right]\right) +$$
$$f\left[8\sqrt{r^6}\right] \times f\left[12\sqrt{r^7}\right] + f\left[10\sqrt{r^7}\right] \times f\left[15\sqrt{r^8}\right],$$
$$f\left[3\sqrt{r^4}\right]^2 + f\left[6\sqrt{r^5}\right]^2 + f\left[9\sqrt{r^6}\right]^2 + f\left[12\sqrt{r^7}\right]^2 + f\left[15\sqrt{r^8}\right]^2,$$
$$f\left[3\sqrt{r^4}\right] \times f\left[4\sqrt{r^5}\right] + f\left[6\sqrt{r^5}\right] \times f\left[8\sqrt{r^6}\right] + f\left[12\sqrt{r^7}\right] \times \left(f\left[9\sqrt{r^6}\right] + f\left[16\sqrt{r^8}\right]\right) +$$
$$f\left[15\sqrt{r^8}\right] \times f\left[20\sqrt{r^9}\right], \ f\left[3\sqrt{r^4}\right] \times f\left[5\sqrt{r^6}\right] + f\left[6\sqrt{r^5}\right] \times f\left[10\sqrt{r^7}\right] +$$
$$f\left[12\sqrt{r^7}\right] \times f\left[20\sqrt{r^9}\right] + f\left[15\sqrt{r^8}\right] \times \left(f\left[9\sqrt{r^6}\right] + f\left[25\sqrt{r^{10}}\right]\right)\Big\},$$
$$\Big\{ f\left[2\sqrt{r^3}\right] \times f\left[8\sqrt{r^6}\right] + f\left[3\sqrt{r^4}\right] \times f\left[12\sqrt{r^7}\right] + f\left[4\sqrt{r^5}\right] \times \left(f\left[\sqrt{r^2}\right] + f\left[16\sqrt{r^8}\right]\right) +$$
$$f\left[5\sqrt{r^6}\right] \times f\left[20\sqrt{r^9}\right], \ f\left[2\sqrt{r^3}\right] \times f\left[4\sqrt{r^5}\right] + f\left[6\sqrt{r^5}\right] \times f\left[12\sqrt{r^7}\right] +$$
$$f\left[8\sqrt{r^6}\right] \times \left(f\left[4\sqrt{r^4}\right] + f\left[16\sqrt{r^8}\right]\right) + f\left[10\sqrt{r^7}\right] \times f\left[20\sqrt{r^9}\right], \ f\left[3\sqrt{r^4}\right] \times f\left[4\sqrt{r^5}\right] +$$
$$f\left[6\sqrt{r^5}\right] \times f\left[8\sqrt{r^6}\right] + f\left[12\sqrt{r^7}\right] \times \left(f\left[9\sqrt{r^6}\right] + f\left[16\sqrt{r^8}\right]\right) + f\left[15\sqrt{r^8}\right] \times f\left[20\sqrt{r^9}\right],$$
$$f\left[4\sqrt{r^5}\right]^2 + f\left[8\sqrt{r^6}\right]^2 + f\left[12\sqrt{r^7}\right]^2 + f\left[16\sqrt{r^8}\right]^2 + f\left[20\sqrt{r^9}\right]^2, \ f\left[4\sqrt{r^5}\right] \times f\left[5\sqrt{r^6}\right] +$$
$$f\left[8\sqrt{r^6}\right] \times f\left[10\sqrt{r^7}\right] + f\left[12\sqrt{r^7}\right] \times f\left[15\sqrt{r^8}\right] + f\left[20\sqrt{r^9}\right] \times \left(f\left[16\sqrt{r^8}\right] + f\left[25\sqrt{r^{10}}\right]\right)\Big\},$$
$$\Big\{ f\left[2\sqrt{r^3}\right] \times f\left[10\sqrt{r^7}\right] + f\left[3\sqrt{r^4}\right] \times f\left[15\sqrt{r^8}\right] + f\left[4\sqrt{r^5}\right] \times f\left[20\sqrt{r^9}\right] +$$
$$f\left[5\sqrt{r^6}\right] \times \left(f\left[\sqrt{r^2}\right] + f\left[25\sqrt{r^{10}}\right]\right), \ f\left[2\sqrt{r^3}\right] \times f\left[5\sqrt{r^6}\right] + f\left[6\sqrt{r^5}\right] \times f\left[15\sqrt{r^8}\right] +$$
$$f\left[8\sqrt{r^6}\right] \times f\left[20\sqrt{r^9}\right] + f\left[10\sqrt{r^7}\right] \times \left(f\left[4\sqrt{r^4}\right] + f\left[25\sqrt{r^{10}}\right]\right),$$
$$f\left[3\sqrt{r^4}\right] \times f\left[5\sqrt{r^6}\right] + f\left[6\sqrt{r^5}\right] \times f\left[10\sqrt{r^7}\right] + f\left[12\sqrt{r^7}\right] \times f\left[20\sqrt{r^9}\right] +$$
$$f\left[15\sqrt{r^8}\right] \times \left(f\left[9\sqrt{r^6}\right] + f\left[25\sqrt{r^{10}}\right]\right), \ f\left[4\sqrt{r^5}\right] \times f\left[5\sqrt{r^6}\right] + f\left[8\sqrt{r^6}\right] \times f\left[10\sqrt{r^7}\right] +$$
$$f\left[12\sqrt{r^7}\right] \times f\left[15\sqrt{r^8}\right] + f\left[20\sqrt{r^9}\right] \times \left(f\left[16\sqrt{r^8}\right] + f\left[25\sqrt{r^{10}}\right]\right),$$
$$f\left[5\sqrt{r^6}\right]^2 + f\left[10\sqrt{r^7}\right]^2 + f\left[15\sqrt{r^8}\right]^2 + f\left[20\sqrt{r^9}\right]^2 + f\left[25\sqrt{r^{10}}\right]^2\Big\}\Big\}\Big\}$$

On the other hand, `TSimplify` will get rid of the square roots and take much less time:

In[ ]:=
```
ClearSystemCache[]
parallelSimplify = AbsoluteTiming[TSimplify[testExpression.testExpression]]
```

Out[ ]=
$\Big\{0.317309, \Big\{\Big\{f[r]^2 + f\big[2\,r^{3/2}\big]^2 + f\big[3\,r^2\big]^2 + f\big[4\,r^{5/2}\big]^2 + f\big[5\,r^3\big]^2,$

$f\big[2\,r^{3/2}\big] \times \big(f[r] + f\big[4\,r^2\big]\big) + f\big[3\,r^2\big] \times f\big[6\,r^{5/2}\big] + f\big[4\,r^{5/2}\big] \times f\big[8\,r^3\big] + f\big[5\,r^3\big] \times f\big[10\,r^{7/2}\big],$

$f\big[2\,r^{3/2}\big] \times f\big[6\,r^{5/2}\big] + f\big[3\,r^2\big] \times \big(f[r] + f\big[9\,r^3\big]\big) + f\big[4\,r^{5/2}\big] \times f\big[12\,r^{7/2}\big] + f\big[5\,r^3\big] \times f\big[15\,r^4\big],$

$f\big[2\,r^{3/2}\big] \times f\big[8\,r^3\big] + f\big[3\,r^2\big] \times f\big[12\,r^{7/2}\big] + f\big[4\,r^{5/2}\big] \times \big(f[r] + f\big[16\,r^4\big]\big) + f\big[5\,r^3\big] \times f\big[20\,r^{9/2}\big],$

$f\big[2\,r^{3/2}\big] \times f\big[10\,r^{7/2}\big] + f\big[3\,r^2\big] \times f\big[15\,r^4\big] +$
$f\big[4\,r^{5/2}\big] \times f\big[20\,r^{9/2}\big] + f\big[5\,r^3\big] \times \big(f[r] + f\big[25\,r^5\big]\big)\Big\},$

$\Big\{f\big[2\,r^{3/2}\big] \times \big(f[r] + f\big[4\,r^2\big]\big) + f\big[3\,r^2\big] \times f\big[6\,r^{5/2}\big] + f\big[4\,r^{5/2}\big] \times f\big[8\,r^3\big] + f\big[5\,r^3\big] \times f\big[10\,r^{7/2}\big],$

$f\big[2\,r^{3/2}\big]^2 + f\big[4\,r^2\big]^2 + f\big[6\,r^{5/2}\big]^2 + f\big[8\,r^3\big]^2 + f\big[10\,r^{7/2}\big]^2,$

$f\big[2\,r^{3/2}\big] \times f\big[3\,r^2\big] + f\big[6\,r^{5/2}\big] \times \big(f\big[4\,r^2\big] + f\big[9\,r^3\big]\big) + f\big[8\,r^3\big] \times f\big[12\,r^{7/2}\big] +$
$f\big[10\,r^{7/2}\big] \times f\big[15\,r^4\big], f\big[2\,r^{3/2}\big] \times f\big[4\,r^{5/2}\big] + f\big[6\,r^{5/2}\big] \times f\big[12\,r^{7/2}\big] +$
$f\big[8\,r^3\big] \times \big(f\big[4\,r^2\big] + f\big[16\,r^4\big]\big) + f\big[10\,r^{7/2}\big] \times f\big[20\,r^{9/2}\big], f\big[2\,r^{3/2}\big] \times f\big[5\,r^3\big] +$
$f\big[6\,r^{5/2}\big] \times f\big[15\,r^4\big] + f\big[8\,r^3\big] \times f\big[20\,r^{9/2}\big] + f\big[10\,r^{7/2}\big] \times \big(f\big[4\,r^2\big] + f\big[25\,r^5\big]\big)\Big\},$

$\Big\{f\big[2\,r^{3/2}\big] \times f\big[6\,r^{5/2}\big] + f\big[3\,r^2\big] \times \big(f[r] + f\big[9\,r^3\big]\big) + f\big[4\,r^{5/2}\big] \times f\big[12\,r^{7/2}\big] + f\big[5\,r^3\big] \times f\big[15\,r^4\big],$

$f\big[2\,r^{3/2}\big] \times f\big[3\,r^2\big] + f\big[6\,r^{5/2}\big] \times \big(f\big[4\,r^2\big] + f\big[9\,r^3\big]\big) + f\big[8\,r^3\big] \times f\big[12\,r^{7/2}\big] +$
$f\big[10\,r^{7/2}\big] \times f\big[15\,r^4\big], f\big[3\,r^2\big]^2 + f\big[6\,r^{5/2}\big]^2 + f\big[9\,r^3\big]^2 + f\big[12\,r^{7/2}\big]^2 + f\big[15\,r^4\big]^2,$

$f\big[3\,r^2\big] \times f\big[4\,r^{5/2}\big] + f\big[6\,r^{5/2}\big] \times f\big[8\,r^3\big] + f\big[12\,r^{7/2}\big] \times \big(f\big[9\,r^3\big] + f\big[16\,r^4\big]\big) +$
$f\big[15\,r^4\big] \times f\big[20\,r^{9/2}\big], f\big[3\,r^2\big] \times f\big[5\,r^3\big] + f\big[6\,r^{5/2}\big] \times f\big[10\,r^{7/2}\big] +$
$f\big[12\,r^{7/2}\big] \times f\big[20\,r^{9/2}\big] + f\big[15\,r^4\big] \times \big(f\big[9\,r^3\big] + f\big[25\,r^5\big]\big)\Big\},$

$\Big\{f\big[2\,r^{3/2}\big] \times f\big[8\,r^3\big] + f\big[3\,r^2\big] \times f\big[12\,r^{7/2}\big] + f\big[4\,r^{5/2}\big] \times \big(f[r] + f\big[16\,r^4\big]\big) + f\big[5\,r^3\big] \times f\big[20\,r^{9/2}\big],$

$f\big[2\,r^{3/2}\big] \times f\big[4\,r^{5/2}\big] + f\big[6\,r^{5/2}\big] \times f\big[12\,r^{7/2}\big] + f\big[8\,r^3\big] \times \big(f\big[4\,r^2\big] + f\big[16\,r^4\big]\big) +$
$f\big[10\,r^{7/2}\big] \times f\big[20\,r^{9/2}\big], f\big[3\,r^2\big] \times f\big[4\,r^{5/2}\big] + f\big[6\,r^{5/2}\big] \times f\big[8\,r^3\big] +$
$f\big[12\,r^{7/2}\big] \times \big(f\big[9\,r^3\big] + f\big[16\,r^4\big]\big) + f\big[15\,r^4\big] \times f\big[20\,r^{9/2}\big],$

$f\big[4\,r^{5/2}\big]^2 + f\big[8\,r^3\big]^2 + f\big[12\,r^{7/2}\big]^2 + f\big[16\,r^4\big]^2 + f\big[20\,r^{9/2}\big]^2, f\big[4\,r^{5/2}\big] \times f\big[5\,r^3\big] +$
$f\big[8\,r^3\big] \times f\big[10\,r^{7/2}\big] + f\big[12\,r^{7/2}\big] \times f\big[15\,r^4\big] + f\big[20\,r^{9/2}\big] \times \big(f\big[16\,r^4\big] + f\big[25\,r^5\big]\big)\Big\},$

$\Big\{f\big[2\,r^{3/2}\big] \times f\big[10\,r^{7/2}\big] + f\big[3\,r^2\big] \times f\big[15\,r^4\big] + f\big[4\,r^{5/2}\big] \times f\big[20\,r^{9/2}\big] +$
$f\big[5\,r^3\big] \times \big(f[r] + f\big[25\,r^5\big]\big), f\big[2\,r^{3/2}\big] \times f\big[5\,r^3\big] + f\big[6\,r^{5/2}\big] \times f\big[15\,r^4\big] + f\big[8\,r^3\big] \times f\big[20\,r^{9/2}\big] +$
$f\big[10\,r^{7/2}\big] \times \big(f\big[4\,r^2\big] + f\big[25\,r^5\big]\big), f\big[3\,r^2\big] \times f\big[5\,r^3\big] + f\big[6\,r^{5/2}\big] \times f\big[10\,r^{7/2}\big] +$
$f\big[12\,r^{7/2}\big] \times f\big[20\,r^{9/2}\big] + f\big[15\,r^4\big] \times \big(f\big[9\,r^3\big] + f\big[25\,r^5\big]\big), f\big[4\,r^{5/2}\big] \times f\big[5\,r^3\big] +$
$f\big[8\,r^3\big] \times f\big[10\,r^{7/2}\big] + f\big[12\,r^{7/2}\big] \times f\big[15\,r^4\big] + f\big[20\,r^{9/2}\big] \times \big(f\big[16\,r^4\big] + f\big[25\,r^5\big]\big),$

$f\big[5\,r^3\big]^2 + f\big[10\,r^{7/2}\big]^2 + f\big[15\,r^4\big]^2 + f\big[20\,r^{9/2}\big]^2 + f\big[25\,r^5\big]^2\Big\}\Big\}\Big\}$

On my computer, parallelization provided a speedup by a factor of:

```
In[ ]:=   notParallelSimplify[[1]] / parallelSimplify[[1]]
```

```
Out[ ]=   4.47438
```

Additional information

Version history

For the full version history, please see the file `CHANGELOG.md` in the GitHub repository.

Feature requests and bug reports

This package is under continuous and active development. If you encounter any bugs, or if you would like to request any additional features, please feel free to open a new issue on GitHub and I will look into it as soon as I can.

Copyright and citing

Copyright (c) 2021 Barak Shoshany (http://baraksh.com) (baraksh@gmail.com). Licensed under the MIT license; please see the file `LICENSE.txt` in the GitHub repository for more information.

Citing information can always be obtained by executing the command `TCite[]`:

```
In[ ]:=   ? TCite
```

```
Out[ ]=   Symbol

          TCite[] displays information on how to cite this package in published research. Thank you for citing my work! :)

          ⌄
```

```
In[ ]:=   TCite[]
```

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

- Shoshany, B., (2021). OGRe: An Object-Oriented General Relativity Package for Mathematica. Journal of Open Source Software, 6(65), 3416, https://doi.org/10.21105/joss.03416

You can also use the following BibTeX entry:

```
@article{Shoshany2021_OGRe,
    author   = {Barak Shoshany},
    doi      = {10.21105/joss.03416},
    journal  = {Journal of Open Source Software},
    number   = {65},
    pages    = {3416},
    publisher = {The Open Journal},
```

```
    title    = {OGRe: An Object-Oriented General Relativity Package for Mathematica},
    url      = {https://doi.org/10.21105/joss.03416},
    volume   = {6},
    year     = {2021}
}
```

Thank you for citing my work! :)

Acknowledgements