Using Cadabra for tensor computations in General Relativity.

Leo Brewin

School of Mathematical Sciences
Monash University, 3800
Australia

10-Sep-2019

Abstract

Cadabra is an open access program ideally suited to complex tensor commutations in General Relativity. Tensor expressions are written in LaTeX while an enhanced version of Python is used to control the computations. This tutorial assumes no prior knowledge of Cadabra. It consists of a series of examples covering a range of topics from basic syntax such as declarations, functions, program control, component computations, input and output through to complete computations including a derivation of the BSSN equations from the ADM equations. Numerous exercises are included along with complete solutions. All of the source code for the examples, exercises and solutions are available on GitHub.

Introduction

The main goal in writing this tutorial was to provide the reader with sufficient knowledge so that they can use Cadabra to do meaningful computations in general relativity. It was written for readers with no prior knowledge of Cadabra and is presented as a series of examples using familiar computations (such as verifying that the Levi-Civita connection is a metric connection) as vehicles to present the various elements of Cadabra.

The tutorial contains many exercises (with complete solutions) that allow the reader to test their understanding as well to explore some of the side issues raised in the main thread of the tutorial.

The LaTeX and Cadabra sources for the tutorial can be found on the author's GitHib site (see Part 4 for the relevant URL).

This tutorial is a significantly extended version of a similar tutorial written in 2009 [1]. Though the 2009 tutorial has been updated to comply with the version 2.0 syntax it does not contain any of the extensive additions introduced in version 2.0. It should be noted that version 1.0 of Cadabra is no longer supported and all users are encouraged to migrate to version 2.0. Note also the version 2.0 syntax is not backward compatible with that of version 1.0.

Computations in General Relativity

Here are three examples of the kinds of computation that are often required in General Relativity.

Numerical computation.

Use a numerical method to evolve the time symmetric initial data for a geodesic slicing of a Schwarzschild spacetime in an isotropic gauge.

Algebraic computation.

Compute the Riemann tensor for the metric $ds^2 = \Phi(r)^2\,(dr^2 + r^2 d\Omega^2)$.

Tensor computations.

Verify that $0 = g_{ab;c}$ given $\Gamma^{a}{}_{bc} = \frac{1}{2} g^{ad}\,(g_{dc,b} + g_{bd,c} - g_{bc,d})$.

What tools are available to perform these computations? For the first example, it is hard to envisage *not* using a computer to do the job. The second example is one which could easily be done by hand or on a computer (using, for example, GRTensorIII [2]). The third examples is so simple that most people would use traditional pencil and paper methods. However, there are many other tensor computations in General Relativity that are particular tedious to push through by hand (e.g., developing higher order Riemann normal expansions of the metric or performing perturbation expansions of the vacuum field equations). So there is very good reason to seek help by way of a computer program designed specially to manipulate tensor expressions. This article will provide a brief introduction to one such program, Cadabra, and how it can be used in General Relativity.

Besides Cadabra, there are a number of other programs that, to varying degrees, can manipulate tensor expressions, including GRTensorIII [2], MathTensor [3], Canon [4], Riemann [5], xAct [6] No attempt will be made here to provide even a cursory review of the above programs (however, see the recent review by MacCallum [7]). Instead, the intention in this tutorial will be to show how Cadabra can be used to do useful work in General Relativity.

Given that Cadabra is just one of a number of programs that can manipulate tensor expressions, the obvious question would be – why chose Cadabra?

One of Cadabra's main selling points is its elegant and simple syntax. This is based on a subset of LaTeX to express tensor expressions, Python to coordinate the computations and some unique Cadabra syntax to describe properties of various objects (e.g., index sets, symmetries, commutation rules etc.). This leads to a shallow learning curve and codes that are clear and easy to read. The core program of Cadabra is written in `C++` including highly optimised procedures for simplifying complex tensor expressions. It has a strong user base, active discussion forums and is under active development.

Another strong point of Cadabra is its use of LaTeX for tensor expressions for not only *input to* Cadabra but also for *output from* Cadabra. This means that output from one Cadabra code can be easily used in other Cadabra codes or even in separate LaTeX documents. Indeed this document is a case in point – all of the results appearing later in this document were computed in separate Cadabra codes and included, without change, from the corresponding Cadabra output.

The following examples were deliberately constructed so as to require little mathematical development (for the current audience) while being of sufficient complexity to allow Cadabra's features to be properly showcased. For the majority of this tutorial no assumptions will be made about the dimensionality of the space other than in Example 10 (4 dimensions) and Example 13 (3 dimensions). No assumptions will be made about the metric (other than it being non-singular) while the connection will be assumed to be metric compatible (i.e., the Levi-Civita connection). Abstract index notation will be used but on the odd occasion where an explicit component based equation is given, the components will be given in a coordinate basis.

The Cadabra software

The full source for Cadabra can be found on the GitHub page [8] while binaries for popular versions of Linux and Windows can be found in the downloads section of the Cadabra home page [9]. There are no binaries for macOS but it is a simple matter to compile from the source using Homebrew. Complete instructions are available on the Cadabra GitHub page [8].

There are two main ways to run Cadabra, either through the command line or through a GUI interface similar to the notebook interfaces used by Jupyter and Mathematica. The command line version of Cadabra works with plain text files, such as `foo.cdb`. These files can be created using any text editor and contain Cadabra statements. To run Cadabra on the file `foo.cdb` you need only type

```
cadabra2 foo.cdb
```

on the command line. In contrast, files like `foo.cnb` are Cadabra notebooks and can contain not only Cadabra statements but also Cadabra output as well as LaTeX markup. These files are not intended to be edited in a plain text editor but rather are created, edited and executed entirely from within the Cadabra GUI. To initiate the GUI and load the notebook `foo.cnb` type

```
cadabra2-gtk foo.cnb
```

on the command line. Once the GUI has started you can edit or execute the current notebook or use the `File` menu item to navigate to other notebooks. The menu in the GUI contains the usual set of entries that should need little explanation. However, if it is not obvious what a particular menu item does, then just click on that item and note what happens – though do chose to work on a test file. There are not many menu items so this click and observe method should reveal most of the menu actions in a short time.

Part 1 Essential elements

This first part of the tutorial consists of a set of examples that are intended for readers with little or no experience with Cadabra. Each section is built around a simple example based on some familiar elements of general relativity. These context based examples are used to introduce the essential elements of Cadabra required for routine tensor computations.

The second part of this tutorial switches the focus from introducing Cadabra to applying Cadabra to more substantial questions (which once again cover well known topics in general relativity). The examples are a hotchpotch reflecting the research interest of the author.

The examples are supported by many exercises with full solutions. The exercises are not essential for progression through the later examples but they do help the reader to test their understanding of basic aspects of Cadabra. They also explore aspects of Cadabra not otherwise covered in the main thread of this tutorial.

Notation

The following examples will contain lines of Cadabra code as well as the corresponding output. The question here is – how is that correspondence conveyed to the reader? The device used here will be to match the output against the line number of the code.

Here is a small fragment of a larger Cadabra code (drawn from Example 5).

```
1   expr := A_{a} v^{a} + B_{a} v^{a} + C_{a} v^{a};
2   zoom       (expr, $B_{a} Q??$)
3   substitute (expr, $v^{a} -> w^{a}$);
4   unzoom     (expr)
```

For the moment try to ignore the code and focus instead on the small line numbers in the left hand margin. These numbers are not part of the Cadabra syntax but have been added here so that individual lines of code can be identified. They are also used as tags to match against the Cadabra output which, in this case, just happens to be (have faith)

$$\begin{aligned} A_a v^a + B_a v^a + C_a v^a &= \ldots + B_a v^a + \ldots && /2/ \\ &= \ldots + B_a w^a + \ldots && /3/ \\ &= A_a v^a + B_a w^a + C_a v^a && /4/ \end{aligned}$$

The weird looking equation numbers on the right hand side are matched to the line numbers in the Cadabra code. Thus /2/ is the output generated by line 2 of the Cadabra code, likewise the output for line 3 is given by /3/.

1 Hello metric connection

How might Cadabra be used to verify that $0 = g_{ab;c}$ given ${\Gamma^{a}}_{bc} = \frac{1}{2}g^{ad}\left(g_{dc,b} + g_{bd,c} - g_{bc,d}\right)$?

This may seem an odd way to start but here is the full Cadabra code.

```
 1  # Define some properties
 2
 3  {a,b,c,d,e,f,h,i,j,k,l,m,n,o,p,q,r,s,t,u#}::Indices.
 4
 5  g_{a b}::Metric.
 6  g_{a}^{b}::KroneckerDelta.
 7
 8  \partial{#}::PartialDerivative.
 9
10  # Define a rule for the Christoffel symbol
11
12  Gamma := \Gamma^{a}_{b c} -> (1/2) g^{a d} (  \partial_{b}{g_{d c}}
13                                              + \partial_{c}{g_{b d}}
14                                              - \partial_{d}{g_{b c}} );
15
16  # Define the covariant derivative of the metric
17
18  cderiv := \partial_{c}{g_{a b}} - g_{a d}\Gamma^{d}_{b c}
19                                  - g_{d b}\Gamma^{d}_{a c};
20
21  # Do the computations
22
23  substitute           (cderiv, Gamma);
24  distribute           (cderiv);
25  eliminate_metric     (cderiv);
26  eliminate_kronecker  (cderiv);
27  canonicalise         (cderiv);
```

The output from the above code is

$$
{\Gamma^{a}}_{bc} \rightarrow \frac{1}{2}g^{ad}\left(\partial_{b}g_{dc} + \partial_{c}g_{bd} - \partial_{d}g_{bc}\right) \qquad /12/
$$

$$
\begin{aligned}
g_{ab;c} &= \partial_{c}g_{ab} - g_{ad}{\Gamma^{d}}_{bc} - g_{db}{\Gamma^{d}}_{ac} && /18/\\
&= \partial_{c}g_{ab} - \frac{1}{2}g_{ad}g^{de}\left(\partial_{b}g_{ec} + \partial_{c}g_{be} - \partial_{e}g_{bc}\right) - \frac{1}{2}g_{db}g^{de}\left(\partial_{a}g_{ec} + \partial_{c}g_{ae} - \partial_{e}g_{ac}\right) && /23/\\
&= \partial_{c}g_{ab} - \frac{1}{2}g_{ad}g^{de}\partial_{b}g_{ec} - \frac{1}{2}g_{ad}g^{de}\partial_{c}g_{be} + \frac{1}{2}g_{ad}g^{de}\partial_{e}g_{bc} - \frac{1}{2}g_{db}g^{de}\partial_{a}g_{ec} - \frac{1}{2}g_{db}g^{de}\partial_{c}g_{ae}\\
&\quad + \frac{1}{2}g_{db}g^{de}\partial_{e}g_{ac} && /24/\\
&= \partial_{c}g_{ab} - \frac{1}{2}{g_{a}}^{e}\partial_{b}g_{ec} - \frac{1}{2}{g_{a}}^{e}\partial_{c}g_{be} + \frac{1}{2}{g_{a}}^{e}\partial_{e}g_{bc} - \frac{1}{2}{g_{b}}^{e}\partial_{a}g_{ec} - \frac{1}{2}{g_{b}}^{e}\partial_{c}g_{ae} + \frac{1}{2}{g_{b}}^{e}\partial_{e}g_{ac} && /25/\\
&= \frac{1}{2}\partial_{c}g_{ab} - \frac{1}{2}\partial_{c}g_{ba} && /26/\\
&= 0 && /27/
\end{aligned}
$$

Each of these line shows selected stages of processing by Cadabra. The zero in the final line shows that $g_{ab;c}$ is indeed zero for the given choice of $\Gamma^{a}{}_{bc}$.

Note that the only part of the above output that was written by Cadabra is the part between the equals sign and the (apparent) equation number on the far right. Everything else was added by the author to put the Cadabra output into context. The number on the far right matches the line number in the source while the text to the left of the equals sign identifies the object associated with the Cadabra output. So though the above output is not exactly what would be seen in the GUI it is important to note that the Cadabra output has not been modified in any way other than to be sandwiched between the equals sign on the left and the line number on the right.

Looking back at the above code, the obvious question is – what does each line do? For some lines the answer is clear but for others there are elements of the syntax that do require further explanation. Thus at this point it is useful to spend a bit of time working through the above Cadabra code in some detail.

Statements in the Cadabra grammar fall into a number of distinct categories: *comments*, *properties*, *expressions*, *algorithms* and a broad category that consists of any valid Python statement. Comments in Cadabra are single lines that begin with one or more spaces (or tabs) followed by the `#` character. Any text after the `#` will be treated as a comment. There are four comments in the above example (lines 1, 10, 16, 21). The statements in lines 3 to 8 assign *properties* to some symbols, while those in lines 12 to 19 define two *expressions* named `Gamma` and `cderiv`. The remaining statements apply *algorithms* to the expressions (i.e., they perform the computations). Note that *algorithms* are, in the eyes of Python, ordinary Python functions. Python functions can also be applied to Cadabra objects and thus could also be described as algorithms. But as this may lead to some confusion the convention adopted in this tutorial is that the term *algorithm* will be reserved exclusively for Cadabra's own functions.

Cadabra statements can consist of one or more lines of text. Thus Cadabra sets clear rules about how a statement can be constructed from a series of lines. It will read its input, line by line, while also looking for a clear marker to indicate the end of the current statement. For *properties* and *expressions* the statement will be terminated by either a dot `.` or a semi-colon `;`. The situation is slightly different for *algorithms* – they are terminated either by a dot, a semi-colon or by the closing right parenthesis of the algorithm. In all cases, Cadabra will generate output only for those statements that end with a semi-colon. Python statements are terminated in the normal Python manner.

Once Cadabra has digested the source it will pass a slightly modified copy onto its own internal version of Python (enhanced to support Cadabra's algorithms). Thus the original Cadabra source must conform to Python's strict (but simple) indentation rules.

What do these statements actually mean? The first statement

```
3   {a,b,c,d,e,f,h,i,j,k,l,m,n,o,p,q,r,s,t,u#}::Indices.
```

simply declares a set of symbols that may be used as indices. The last symbol `u#` informs Cadabra that an infinite set of indices of the form `u1,u2,u3`$\cdots$ is allowed. If you prefer to work with Greek indices then you could declare

```
{\alpha,\beta,\gamma,\mu,\nu,\theta,\phi#}::Indices.
```

Note that all of the usual LaTeX Greek symbols are understood by Cadabra. They can be used as indices or symbols (e.g., `\Gamma` can be used to denote a Christoffel symbol).

The next pair of statements

```
5   g_{a b}::Metric.
6   g_{a}^{b}::KroneckerDelta.
```

declares that `g_{a b}` represents a (symmetric) metric and that ${g_a}^b$ is the usual Kronecker delta (i.e., ${g_a}^b = \delta^b_a$).

The fourth statement

```
8   \partial{#}::PartialDerivative.
```

assigns to the symbol `\partial` a derivative property. Note that the `#` in `\partial{#}` signifies that any number of indices (up or down) are allowed. That is both `\partial{a}` and `\partial{a b c d}` will be seen by Cadabra as derivative operators. This interpretation of `{#}` carries over to other declarations, for example `\delta{#}::KroneckerDelta` declares `\delta` to be a Kronecker delta with any number of upper or lower indices (and in any order).

The next two statements define two expressions, `Gamma` and `cderiv`.

```
10  # Define a rule for the Christoffel symbol
11
12  Gamma := \Gamma^{a}_{b c} -> (1/2) g^{a d} (  \partial_{b}{g_{d c}}
13                                              + \partial_{c}{g_{b d}}
14                                              - \partial_{d}{g_{b c}} );
15
16  # Define the covariant derivative of the metric
17
18  cderiv := \partial_{c}{g_{a b}} - g_{a d}\Gamma^{d}_{b c}
19                                  - g_{d b}\Gamma^{d}_{a c};
```

The name of the expression appears to the left of the '`:=`' characters while the corresponding tensor expression appears on the right using a familiar LaTeX syntax. Tensor indices such as `a,b,c...` should always be separated by one or more spaces (unlike the case in LaTeX). This ensures that Cadabra knows exactly how many indices belong to an object (e.g., `g_{ab}` would be interpreted as an object with *one* covariant index `ab`). This rule can be relaxed when the index set contains its own delimiter such as the slash that appears when indices are written using LaTeX names. Thus an object like `g_{\alpha\beta}` clearly contains just two indices.

Note carefully the braces around the metric term in `\partial_{c}{g_{a b}}`. This is essential – the symbol `\partial` is an operator and thus needs an argument to act on, namely, the argument contained inside the pair of braces.

There is one very important operational difference between the pair of expressions `cderiv` and `Gamma`. The expression `cderiv` defines a Cadabra object that will be manipulated in stages towards the final result (in line 27). These changes are obtained by applying Cadabra's algorithms (lines 23 to 27) to `cderiv`. The other expression, `Gamma`, defines a substitution rule that informs Cadabra how to replace any instance of ${\Gamma^a}_{bc}$ with the appropriate combination of the metric and its derivatives.

Though Cadabra now knows how to make that substitution it will hold fire until told do so – which in this example just happens to be the very next statement, the `substitute` algorithm in line 23. This ability to defer the application of substitution rules at the users discretion is one of Cadabra's main features. In other algebraic software, such as Maple and Mathematica, substitutions are propagated forward from the point at which they are declared. They will also be imposed indirectly on existing expressions. Cadabra, in contrast, allows the user to choose when a rule should be applied as well as on which expressions the rule will be applied to. This gives the user considerable freedom in developing a strategy to achieve a desired computational goal.

The upshot is that after Cadabra has executed the `substitute` algorithm, the object `cderiv` will consist solely of terms built from the metric and its derivatives. Though this may look simple there is a very important and subtle detail that must be noted. The substitution rule `Gamma` as given above was for $\Gamma^{a}{}_{bc}$ yet the expression for `cderiv` requires $\Gamma^{d}{}_{bc}$ and $\Gamma^{d}{}_{ac}$. Cadabra handles this index manipulation by relabelling dummy indices in such a way as to avoid index clashes. This feature also exists in MathTensor and xAct.

The remaining few statements

```
distribute          (cderiv);
eliminate_metric    (cderiv);
eliminate_kronecker (cderiv);
canonicalise        (cderiv);
```

serve only to massage the expression towards the expected result – zero. Each of the statements applies an algorithm to the expression `cderiv` with the result replacing the original value of `cderiv`. That is, Cadabra's algorithms makes in-place changes to Cadabra objects. The algorithm `distribute` is used to expand products, it will expand `a (b+c)` into `a b + a c`. In line 5 of the code the property `::Metric` was given to `g_{a b}`. This is used by the `eliminate_metric` algorithm to convert combinations such as `g_{a c} g^{c b}` into a Kronecker-delta δ_{a}^{b} which (not surprisingly) is eliminated by the `eliminate_kronecker` algorithm. The `canonicalise` algorithm is one of Cadabra's most useful algorithms (on a par with `substitute`) as it can apply a wide range of simplifications and general housekeeping. In this case it makes use of the symmetric property of the metric to complete the final step of the calculation. The result in line /27/ is zero as expected.

1.1 Cadabra syntax summary

The above discussion has introduced some key elements of the Cadabra syntax. Other elements will be discussed later as the need arises. Though this does present a shallow learning curve (consider the alternative where mastery of the full syntax tree is required before seeing any examples) it does mean that important information is scattered throughout the tutorial. This of course makes it harder to find key information after the first reading. To mitigate that problem, here is a short summary of the Cadabra syntax that will be seen in later examples and exercises.

This summary will only cover the very basics needed to work through this tutorial. Many elements of the Cadabra syntax will not be discussed here. For a complete and definitive reference please see the Cadabra web pages https://cadabra.science/help.html.

The first point to emphasise is that Cadabra is built upon Python and LaTeX and thus Cadabra codes must adhere to their respective syntaxes.

Parsing

Parsing a Cadabra program serves two purposes. First, it checks for correctness of the code.

Second, it converts any statements unique to Cadabra (such as `{a,b,c}::Indices`) into statements that can be understood by Python. The result is a new program written entirely in Python (with the Cadabra elements implemented as function calls to an external library). This preprocess step can be seen in action using the command line tool `cadabra2python`. To create the Python code for the file `foo.cdb` you need only type

```
cadabra2python foo.cdb foo.py
```

Statement termination

Since statements can be composed of one or more lines of text there must be some rule for deciding when a series of lines constitutes a single statement. Python statements are terminated according to Python's rules. Here are some examples of valid and invalid Python statements.

```
foo = bah                   # valid
foo = simplify (bah)        # valid
bah = truncate (foo,3);     # invalid, ; not allowed here
bah := derive (foo)         # invalid, use = not := for Python assignment
```

A Cadabra statement can be terminated using either a dot `.`, a semi-colon `;` or the closing right parenthesis `)` for functions and algorithms. Using a semi-colon to terminate a statement will force Cadabra to print the output generated by the statement. Here are two Cadabra statements, only the first is valid.

```
foo := A_{a} B_{b};         # valid
bah  = B_{a} A_{b}.         # invalid, use := not = for Cadabra assignment
```

Continuation

Python statements can be split across lines in a number of ways including line breaks between items in a list. A slash at the end of line also signifies a continuation. This is standard Python. For Cadabra the rules are not so simple. Properties (e.g., `::Indices`) can *not* be split across multiple lines. However, multiple instances are allowed and will be stored as a sequence of property lists. Examples of this will be seen later in Exercise 1.6 and Example 12. In contrast, Cadabra expressions such as `foo := A_{a} B_{b}.` *can* be split across more than one line by including line breaks as needed and with proper termination (e.g., a dot or a semi-colon). Note that Python's indentation rules apply only to the first line of a group – the remaining lines can be indented to suit. See also the discussion on very long lines in the Miscellaneous section of Part 4.

Identifiers

Identifiers can be built using standard alphanumeric characters (excluding the special characters like `!@#$%^$` etc.). Python allows underscore characters but as they are also used by LaTeX to introduce subscripts it is best to *not* use an underscore in a Cadabra identifier (it is allowed but it can cause confusion for the reader). Cadabra also allows the standard LaTeX symbol names (e.g.,`\Gamma`) to be used as identifiers. In this tutorial all identifiers will be built from the alphanumeric characters (a to z, A to Z and 0 to 9) and occasionally LaTeX symbol names.

Assignment

Assignments in Python are made using `=` as in `foo = "abc"` while in Cadabra they are made (mostly) using `:=`. One reason for this small difference is the simple fact that Python does not understand assignments made from LaTeX expressions. For example, `foo = A_{a} B_{b}` would make no sense in pure Python. Thus `:=` is used to signal that the assignment `foo := A_{a} B_{b}.` must be made by Cadabra rather than Python.

The same assignment can also be made using Cadabra's `Ex` function using `foo = Ex(r"A_{a} B_{b}")`. This function takes a (raw) string, converts it into a Python compliant datastructure (an `Exnode`) and assigns the result to the left hand side (i.e., to `foo`). Since this statement is handled by Cadabra's own enhanced version of Python (to include `Ex`) the assignment uses `=` rather than `:=`. Note also the absence of an explicit termination character (no dot or colon) and also the use of the raw string `r"..."`. The raw string is not needed in this example but would be required if the string contained any slashes (e.g., a LaTeX symbol like `\Gamma`). The function `Ex` is very useful when building expressions from smaller pieces (see for example the function `truncate` in Example 4).

It must be noted that identifiers carry now residual information about their origins. Once an identifier is created (using `=` or `:=`) it carries no residual information about its creator (`=` or `:=`). Thus any statement like `bah = foo` will have the usual Python meaning, namely, that `bah` and `foo` share one copy of the data pointed to by `foo`. There are many occasions were a second distinct copy of the data is required. Copies of pure Cadabra objects (i.e., created using `:=` or `Ex`) can be made using statements like `bah := @(foo);`. The `@(...)` is simply a function that returns a copy of the given Cadabra object. This construction will be used many times in this tutorial (the first instance can be seen in Exercise 1.7).

Comment character

The hash character `#` is used in Python to start a comment. All text on the line following and including the hash will be ignored by Python. But in Cadabra the hash character is used in many property declarations. Here are some examples (put aside for the moment what these mean, just accept that they are valid Cadabra statements)

```
{a,b,c,d,e#}::Indices.
\delta{#}::KroneckerDelta.
D{#}::LaTeXForm{"\nabla"}.
```

This dual use of the hash character forces a compromise to be made – comments are not allowed as trailing text on a pure Cadabra line (e.g., on the end of a property declaration). Other comments, for example lines that begin with a hash or as trailing text on a pure Python line, are allowed.

Indentation

All Cadabra programs must conform to Python's indentation rules. These rules may, at first sight, seem strange for people not familiar with Python but they are not too hard to understand. The basic idea is that code blocks that might in other languages be wrapped in `{}` or `begin/end` pairs are indented by at least one space (usually four spaces) from the surrounding code. This applies to if-then-else blocks, for-loops, function definitions and nested blocks (and more). Here are a few examples

```
foo = 123
if foo == 123:
   bah = 456
   print ("in True")
else:
   bah = 789
   print ("in False")
print (bah)
```

```
def swap (my_string):
    first_char   = my_string[0]
    second_char  = my_string[1]
    my_string[0] = second_char
    my_string[1] = first_char
    return my_string
```

Similar indenting is often used in other languages as a way to improve the readability of the code. In Python this use of indentation is mandatory.

CamelCase and snake_case

Though Cadabra is case sensitive it does not stipulate which case to use for various constructions. However, the common practice is to use `CamelCase` for properties (e.g., `::Indices`, `::Derivative`) and `snake_case` for algorithms and functions (e.g., and `sort_product`, `product_rule`). Two obvious (trivial) exceptions are the function `Ex` and the use of uppercase LaTeX names for identifiers such as `\Gamma`.

Line splitting

Cadabra allows *expressions* to be split across one or more lines such as

```
Rabcd := R_{a b c d} ->   \partial_{c}{\Gamma_{a b d}}
                        - \partial_{d}{\Gamma_{a b c}}
                        + \Gamma_{e a d} \Gamma^{e}_{b c}
                        - \Gamma_{e a c} \Gamma^{e}_{b d}.
```

However, it does *not* allow property lists or anonymous rules (i.e., `$...$`) to be split. Thus each of the following statements will raise an error.

```
{\alpha,\beta,\gamme,\delta,
 \mu,\nu,\sigma,\rho,\tau,\theta}::Indices.

{R_{\alpha\beta\gamm\delta},
 \partial_{\mu}{R_{\alpha\beta\gamm\delta}}}::SortOrder.

 substitute (foo, $R -> R_{\mu\nu} g^{\mu\nu},
                   R_{\mu\nu} -> R_{\alpha\mu\beta\nu} g^{\alpha\beta}$)
```

For property lists the preferred solution is to use one line (no matter how long it might be). Thus you would use

```
{\alpha,\beta,\gamme,\delta,\mu,\nu,\sigma,\rho,\tau,\theta}::Indices.

{R_{\alpha\beta\gamm\delta},\partial_{\mu}{R_{\alpha\beta\gamm\delta}}}::SortOrder.
```

See the discussion on line splitting in Part 3 below for an alternative solution.

I think there is a bug when adding rules that contain single elements. See `../cadabra/bugs/lcb11`

But for rules there is a better solution – they can be given names and they can be concatenated. Thus you could write

```
Ricci := R -> R_{\mu\nu} g^{\mu\nu}.
```

```
Riemann := R_{\mu\nu} -> R_{\alpha\mu\beta\nu} g^{\alpha\beta}.
BothRules := Ricci + Riemann.
substitute (foo, BothRules)          # good
substitute (bah, Ricci + Riemann)    # okay too
```

You can also split named rules across lines, for example

```
Ricci := R ->
         R_{\mu\nu} g^{\mu\nu}.
Riemann := R_{\mu\nu} ->
           R_{\alpha\mu\beta\nu} g^{\alpha\beta}.
BothRules =   Ricci \
            + Riemann
substitute (foo, BothRules)
```

Though it must be admitted that this second version is an overkill (in this case). Note also the slash used in the pair of lines used to build `BothRules`. This is standard Python syntax (without the slash Python will report an indentation error).

Exercises

1.1. Given that

$$\Gamma^{a}{}_{bc} = \frac{1}{2} g^{ad} \left(\partial_b g_{dc} + \partial_c g_{bd} - \partial_d g_{bc}\right)$$

use Cadabra to verify that

$$\Gamma^{a}{}_{bc} = \Gamma^{a}{}_{cb}$$

Hint: Define a rule for $\Gamma^{a}{}_{bc}$ based on the above definition. Then apply that rule to the expression $\Gamma^{a}{}_{bc} - \Gamma^{a}{}_{cb}$ and finally use suitable Cadabra algorithms to simplify the result.

1.2. Define Γ_{abc} (the Christoffel symbols of the first kind) by

$$\Gamma_{abc} = g_{ad} \Gamma^{d}{}_{bc}$$

Use Cadabra to verify that

$$\Gamma_{abc} + \Gamma_{bac} = \partial_c g_{ab}$$

Hint: Define two rules, one for $\Gamma^{a}{}_{bc}$ as per the previous exercise and one for Γ_{abc} as per the above definition. Apply both rules to the expression $\Gamma_{abc} + \Gamma_{bac} - \partial_c g_{ab}$ then use suitable Cadabra algorithms to simplify the result.

1.3. Modify your Cadabra code from the previous example to apply just *one* rule to $\Gamma_{abc} + \Gamma_{bac} - \partial_c g_{ab}$.

Hint: Cadabra allows rules to act not only on expressions but also on other rules. Use this feature to construct a single rule from the original pair.

Note. To avoid a Cadabra runtime error you may need to replace `::Indices.` with `::Indices(position=independent).` This point will be discussed in more detail in the following example (on covariant differentiation).

1.4. This exercise is a brief experiment with Cadabra's `sort_product` algorithm. Apply `sort_product` to each of the following expressions and carefully note the result. You should be able to glean from these examples the default sort order used by Cadabra.

$$\begin{array}{ll}
(1) & C^f w^e B^d v^c A^b u^a \\
(2) & \Omega_f \gamma_e \Pi_d \beta_c \Gamma_b \alpha_a \\
(3) & C^f w^e B^d v^c A^b u^a \Omega_f \gamma_e \Pi_d \beta_c \Gamma_b \alpha_a \\
(4) & \partial_f C^f w^l \partial_d B^d v^k \partial_b A^b u^j \Omega_i \partial^e \gamma_e \Pi_h \partial^c \beta_c \Gamma_g \partial^a \alpha_a \\
(5) & \partial C w \partial B v \partial A u \Omega \partial \gamma \Pi \partial \beta \Gamma \partial \alpha \\
(6) & A_b A_a A_{cde} A_{fg} \\
(7) & A_a A^a + A^a A_a
\end{array}$$

The results of the first four examples shows that Cadabra's default sort can be summarised as

UPPERCASE < SLASH-UPPERCASE < slash-lowercase < lowercase

The fifth example shows that Cadabra's default sort ordering usually ignores indices. The exception, as shown in the final pair of examples, is when object names are repeated. In such cases Cadabra will sort the terms based on their indices.

Cadabra does allow some control over the sort order by explicitly listing the order in a `::SortOrder` property. Each of the following are valid instances of a sort order list

```
{F,E,D,C,B,A}::SortOrder.
{R_{a b}, R_{a b c d}, R^{a b c d}}::SortOrder.
{\partial_{a}{g_{b c}}, \partial{a b}{R}}::SortOrder.
```

1.5. Look back at the last example in the previous exercise. Cadabra returned $A_a A^a + A^a A_a$ which, assuming A is self-commuting, can be simplified to $2A_a A^a$. If the original expression had been $A_a Z^a + Z^a A_a$ then the result (after `sort_product`) would have been $2A_a Z^a$. This should give you a clue as to how the first expression (involving just A) can be sorted to give $2A_a A^a$. Write a code that does the job. An extension of this idea will be devloped later in Exercise 4.6.

1.6. Cadabra does allow multiple instances of the `::SortOrder` property. Run the following code through Cadabra and observe the result.

```
{D,C,B,A}::SortOrder.

foo := A B C D;
sort_product (foo);

{V,U}::SortOrder.

foo  := U V A B C D;
sort_product (foo);

{A,B,C,D}::SortOrder.

foo := U V D C B A;
sort_product (foo);
```

The results may seem surprising. The final results for `foo` is `foo = D C B A V U`. But looking at third instance of `SortOrder` it is reasonable to expect `foo = A B C D V U`. How can this be? The answer lies in how Cadabra handles multiple instances of the `SortOrder`. The logic is a bit tricky but it goes as follows. The sorting is done using Bubble Sort. This works by sorting a list one pair at a time. Suppose `P` and `Q` define a pair `PQ`. The correct order might require the pair to be swapped. That decision, to swap or not, is made by first searching for the first list that contains `P`. If that list also contains `Q` then that list will be used to determine if `P`and `Q` should be swapped. In all other cases (i.e., when a suitable `SortOrder` list can not be found) the correct order for `PQ` will be found from Cadabra's default sort order.

The upshot is that repeat entries in `SortOrder`, either in a list or across lists, play no part in setting the order. The repeat entries, such as the entire third list above, will in effect by ignored.

An alternative to using `SortOrder` will be presented later in Exercise 4.6.

1.7. This exercise explores the differences between `foo = bah` and `foo := @(bah)`. The first ensures that both `foo` and `bah` share the same data. Any changes to either `foo` or `bah` will be shared by its partner. Using `foo := @(bah)` creates a fresh copy of `bah` and assigns `foo` to that copy. Any subsequent changes to `foo` will not be reflected in `bah` and vice-versa.

The following code demonstrates this behaviour by using the `id` function to reveal the location in the computer's memory where the object resides (i.e., a memory address). Careful inspection of the source and the corresponding output should convince you that the above description is correct.

```
{a,b,c,d,e,f,h#}::Indices.

foo := B_{b} A_{a}.
bah := A_{a} C_{c}.

print("foo = "+str(foo))
print("bah = "+str(bah)+"\n")

print("type foo = "+str(type(foo)))
print("type bah = "+str(type(bah))+"\n")

print("id foo = "+str(id(foo)))
print("id bah = "+str(id(bah))+"\n")

bah = foo

print("foo = "+str(foo))
print("bah = "+str(bah)+"\n")

sort_product (foo)

print("bah = "+str(bah)+"\n")

print("id foo = "+str(id(foo)))
print("id bah = "+str(id(bah))+"\n")

bah := @(foo).
```

```
print("id foo = "+str(id(foo)))
print("id bah = "+str(id(bah))+"\n")
```

1.8. The following code contains a number of syntax errors. Identify and correct the errors then test the corrected code by running it through Cadabra.

```
{a,b,c,d,e,f#}::Indices.
C{#}::Symmetric.

foo := A_{a} B_{b} + C_{ab}.
bah := B_{b} A_{a} + C_{ba}.
meh := @(foo) - @(bah)

if meh == 0:
   print ("meh is zero, and all is good")
      success = True.
else:
   print ("meh is not zero, oops")
      success = False.

canonicalise (meh).
sort_product (meh);

{\alpha\beta\gamma}::Indices.

foo := Ex ("A_{ab} - A_{a b}");
bah := Ex ("A_{\alpha\beta} - A_{\alpha \beta}");
```

1.9. This and the following two exercises deal with simple index manipulations. Consider a pair of tensors A_a and B_b defined by

$$A_a = A_{ac}C^c \qquad \text{and} \qquad B_b = B_{bc}C^c$$

The two tensors have distinct free indices but share a common dummy index c. How does Cadabra handle the possible index clash when constructing a product of A_a with B_b? The answer can be found by running this simple code

```
{a,b,c,d,e,f,h#}::Indices.

foo := A_{a c} C^{c}.
bah := B_{b c} C^{c}.

foobah := @(foo) @(bah).
```

Run the above code and look closely at the result. You should notice that Cadabra has automatically adjusted the dummy indices to avoid a clash.

1.10. Another common index operation is to relabel the free indices. Write a Cadabra code that relabels A_{abc} to A_{uvw}. You can do this by contracting A_{abc} with suitably chosen Kronecker deltas.

1.11. Suppose now that you need to cycle the free indices, say from A_{abc} to A_{bca}. This can be done using two rounds of Kronecker deltas. But there is an elegant and simpler solution using Cadabra's substitution rules. The idea is to create a rule that replaces a temporary object like T_{abc} with A_{abc}. Then apply that rule (using `susbtitute`) to T_{bca}. Note the cycled indices on T. Write a Cadabra code that implements this neat trick.

---

2 Covariant differentiation

Cadabra does not have native algorithms to compute covariant derivatives, Riemann tensors, Ricci tensors and so on. One of its strengths is that it provides a rich set of simple tools by which such objects can be constructed. This second example will show how Cadabra can be trained to compute covariant derivatives.

For a simple vector such as v^a the standard textbook definition of the covariant derivative $\nabla_b v^a$ is

$$\nabla_b v^a = \partial_b v^a + \Gamma^a{}_{cb} v^c$$

A simple way to implement this in Cadabra would be to first define symbols to represent the derivative operators

```
\nabla{#}::Derivative.
\partial{#}::PartialDerivative.
```

and then define a rule for the actual covariant derivative

```
deriv := \nabla_{a}{v^{b}} -> \partial_{a}{v^{b}} + \Gamma^{b}_{c a} v^{c}.
```

This rule could then be used to replace any instances of $\nabla_b v^a$ in a Cadabra expression such as `foo` with the appropriate partial derivatives and Christofell symbols using

```
substitute (foo,deriv)
```

From here it is a simple matter to construct a working code – just add a definition for the indices and some lines to simplify the output. This leads to the following minimal working code.

```
{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u#}::Indices.

\nabla{#}::Derivative.
\partial{#}::PartialDerivative.

# rule for covariant derivative of v^{a}

deriv := \nabla_{a}{v^{b}} -> \partial_{a}{v^{b}} + \Gamma^{b}_{c a} v^{c}.

# create an expression

foo := \nabla_{a}{v^{b}}.

# apply the rule, then simplify

substitute    (foo,deriv)
canonicalise  (foo)
```

The corresponding output is

$$\begin{aligned} \nabla_a v^b &= \partial_a v^b + \Gamma^b{}_{ca} v^c && /16/ \\ &= \partial_a v^b + \Gamma^{bc}{}_a v_c && /17/ \end{aligned}$$

The first line in the output is as expected – it simply repeats the definition given above. However, the second line is not exactly as expected – note how the second index on the Christoffel symbol has been raised (while the corresponding index on v has been lowered). Though this is mathematically correct, it is not standard practice and it would be better if Cadabra could be persuaded to not do such index gymnastics. The solution is to inform Cadabra that the upper and lower indices are to be left as is by adding the qualifier `position=independent` to the `::Indices` property. That is

```
1  {a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u#}::Indices(position=independent).
```

The corresponding output is now

$$\begin{aligned}\nabla_a v^b &= \partial_a v^b + \Gamma^b{}_{ca} v^c && /16/\\ &= \partial_a v^b + \Gamma^b{}_{ca} v^c && /17/\end{aligned}$$

In this instance the changes brought about by specifying `(position=independent)` are simply cosmetic. There are, however, cases where strict control must be maintained over the raising and lowering of indices (usually by explicit use of the metric). This is particularly true for expressions that involve derivative operators. Without the `(position=independent)` qualifier the `canonicalise` algorithm might (incorrectly) raise or lower an index inside the derivative, such as b in $\partial_a V^b$. Of course, if the derivative operator is compatible with the metric (e.g., $\nabla g = 0$) then there is no issue and the indices can be declared without the `(position=independent)` qualifier (though the aesthetics of the output might not be ideal).

The above discussion also explains why `(position=independent)` was required in Exercise 1.3. Without it Cadabra will treat `\Gamma_{a b c}` and `\Gamma^{a}{}_{b c}` as one and the same. Thus any attempt to apply a substitution on `\Gamma^{a}{}_{b c}` in the rule `\Gamma_{a b c} -> g_{a d}\Gamma^{d}{}_{b c}` will actually be applied to both `\Gamma` terms. This removes all trace of `\Gamma` from the rule (you can verify this by making small changes to your code from Exercise 1.3). See also Exercise 2.8 for more adventures with indices.

There remains one minor problem with the above code – the rule in line 8 was designed explicitly for covariant derivatives of v^a and thus is not applicable to other objects such as u^a or expressions like $u^a + v^a$. The solution lies in defining a rule that is applicable to a wider class of objects. Cadabra has a simple syntax that uses a single post-fix question mark to define a generic object. Thus `A?` will match objects such as `P`, `Q`, `PQ` etc. The upshot is that the original rule for the covariant derivative can be generalised to

```
6  # template for covariant derivative of a vector
7
8  deriv := \nabla_{a}{A?^{b}} -> \partial_{a}{A?^{b}} + \Gamma^{b}_{c a} A?^{c}.
```

This rule will work as expected when applied to $\nabla_a u^b$, $\nabla_a v^b$ and $\nabla_a u^b + \nabla_a v^b$.

Exercises

2.1. Use the definitions

$$\nabla_a u^b = \partial_a u^b + \Gamma^b{}_{ca} u^c$$

and

$$\nabla_a v_b = \partial_a v_b - \Gamma^c{}_{ab} v_c$$

to verify that

$$\nabla_a \left(v_b u^b\right) = \partial_a \left(v_b u^b\right)$$

Hint: Begin by applying the product rule to $\nabla_a \left(v_b u^b\right) - \partial_a \left(v_b u^b\right)$. You can do so using either Cadabra's `product_rule` algorithm or you can create two rules, one for each of the derivative operators, ∇ and ∂. You might also need `product_sort` and `rename_dummies` for housekeeping.

2.2. Given A_a and B_b, define v_{ab} by $v_{ab} = A_a B_b$. Adapt your Cadabra codes for $\nabla_a v_b$ to verify that

$$\nabla_a v_{bc} = \partial_a v_{bc} - \Gamma^d{}_{ba} v_{dc} - \Gamma^d{}_{ca} v_{bd}$$

2.3. In a similar vein, given $v^a{}_b = A^a B_b$ show that

$$\nabla_a v^b{}_c = \partial_a v^b{}_c + \Gamma^b{}_{da} v^d{}_c - \Gamma^d{}_{ca} v^b{}_d$$

This and the previous exercise show how easy it is to use Cadabra to verify standard textbook definitions for covariant derivatives. Setting $v^a{}_b = A^a B_b$ might appear to limit the validity of the above result. However, since any tensor can be built as a linear combination of products of vectors and dual-vectors and as the above is linear in $v^a{}_b$ it follows that the result does hold for any choice of $v^a{}_b$ (as expected). This same trick could be used to discover equations for covariant derivatives of any tensor, however, it is much easier to just code up the textbook definition as shown in the following example.

2.4. The objective in this and the following exercise is to build a single rule that expresses $\nabla_a \nabla_b v^c$ in terms of v, Γ and their partial derivatives. As a start, use the following fragment to build a Cadabra code. Observe the result of the call to `substitute`.

```
deriv1 := \nabla_{a}{v^{b}}              -> \partial_{a}{v^{b}}
                                          + \Gamma^{b}_{d a} v^{d}.

deriv2 := \nabla_{a}{\nabla_{b}{v^{c}}} -> \partial_{a}{\nabla_{b}{v^{c}}}
                                          + \Gamma^{c}_{d a} \nabla_{b}{v^{d}}
                                          - \Gamma^{d}_{b a} \nabla_{d}{v^{c}}.

substitute (deriv2,deriv1)
```

2.5. The previous exercise showed that calls to `substitute` will be applied to all terms in an expression, in this case to both sides of the rule `deriv2`. One way to avoid this problem is to ensure that the left hand side of the expression does not contain the target of the rule being applied. Using the same rules as above for `deriv1` and `deriv2` build a new code using

```
expr := v^{c}_{b a} -> \nabla_{a}{\nabla_{b}{v^{c}}}.

substitute (expr,deriv2)
substitute (expr,deriv1)
```

You might like to tidy the final result by substituting $\nabla_a \nabla_b v^c$ for $v^c{}_{ba}$. A variation on this code will be presented in the following section on the Riemann tensor.

2.6. Use Cadabra to verify that for any scalar function ϕ

$$(\nabla_a \nabla_b - \nabla_b \nabla_a)\phi = (\Gamma^c{}_{ab} - \Gamma^c{}_{ba})\partial_c \phi$$

2.7. A popular strategy in proving various theorems in differential geometry is to first assume that coordinates have been chosen so that partial derivatives of the metric vanish at some (arbitrarily) chosen point. This step kills a whole raft of terms and from there the theorem becomes almost trivial to prove. Suppose that this step, of setting $\partial g = 0$, is to be applied to the following expression (this is not part of any deep theorem it was invented just to set the scene)

$$A_{ab} = g_{ab} + \partial_c g_{ab} x^c + \partial_{cd} g_{ab} x^c x^d$$

Write a Cadabra code that uses a substitution rule to set the first partial derivatives to zero. Be sure to kill only the first derivatives.

2.8. Cadabra actually has three choices for the `position` keyword, namely `position=free`, `position=fixed` and `position=independent`. with `position=free` as the default. The difference between the three choices is the degree of freedom given to Cadabra in raising and lowering indices. As already seen, `position=free` allows Cadabra to freely raise and lower indices while `position=indpendent` instructs Cadabra to leave index raising and lowering to the user. The choice `position=fixed` lies between these two extremes. It will allow `canonicalise` to raise and lower matching dummy indices. These three cases are demonstrated in the following code. Run the code and look closely the output. You should see the behaviour just described.

```
{a,b,c}::Indices(position=free).

foo := A_{a b} + A^{a b}.

substitute (foo, $A_{a b} -> B_{a b}$)

{p,q,r}::Indices(position=fixed).

foo := A_{p q} B^{p q} + A^{p q} B_{p q}.

canonicalise (foo)

{u,v,w}::Indices(position=independent).
```

```
foo := A_{u v} B^{u v} + A^{u v} B_{u v}.

canonicalise (foo)
```

Note that mixed indices as in $A_{ab} + A^{ab}$ should never occur in general relativity. Cadabra will flag such cases as an error when using `position=fixed` or `position=independent`.

These exercises show that it is not too hard to create rules for each covariant derivative of interest though it might be tedious listing all possible cases (even when using constructions like `A?` etc.). Unfortunately, Cadabra's pattern matching repertoire, such as `A?`, does not extend to arbitrary tensors. Thus it is not possible to write a single rule that covers every possible form of covariant derivative. However, with Cadabra's native interface to Python, it is possible to write a function that will return the full covariant derivative for an arbitrary tensor. Unfortunately, the inner workings of this function draw upon many aspects of Cadabra's core syntax that are beyond the scope of this tutorial. For full details see https://cadabra.science/notebooks/ref_programming.html

3 To Riemann and beyond

The Riemann tensor for a symmetric connection can be computed (in a coordinate basis) using

$$R^{a}{}_{bcd} = \partial_c \Gamma^{a}{}_{bd} - \partial_d \Gamma^{a}{}_{bc} + \Gamma^{e}{}_{bd}\Gamma^{a}{}_{ce} - \Gamma^{e}{}_{bc}\Gamma^{a}{}_{de}$$

A standard computation in differential geometry then shows that

$$V^{a}{}_{;b;c} - V^{a}{}_{;c;b} = -R^{a}{}_{dbc} V^{d}$$

where the symbol `;` denotes covariant differentiation for the connection $\Gamma^a{}_{bc}$. The purpose of this example is to show how Cadabra can be used to recover the above definition of $R^a{}_{bcd}$ by direct computation of the left hand side of the previous equation.

One way to expand $V^a{}_{;b;c} - V^a{}_{;c;b}$ is combine two expressions, one for $V^a{}_{;b}$ and one for $V^a{}_{b;c}$ with $V^a{}_b$ equal to $V^a{}_{;b}$. This suggest the following Cadabra fragment

```
# rules for the first two covariant derivs of V^a

deriv1 := V^{a}_{; b}      -> \partial_{b}{V^{a}}
                             + \Gamma^{a}_{c b} V^{c}.

deriv2 := V^{a}_{; b ; c}  -> \partial_{c}{V^{a}_{; b}}
                             + \Gamma^{a}_{d c} V^{d}_{; b}
                             - \Gamma^{d}_{b c} V^{a}_{; d}.
```

Though this is a faithful transcription of the underlying mathematics this fragment is taking a small liberty with the syntax – Cadabra might treat the `;` as a tensor index despite not being declared in the list of valid indices (i.e., the `::Indices`). It turns out that Cadabra is smart enough to not make this mistake, either by good design or by good fortune. However, any ambiguity (on Cadabra's part) can be avoided by using

```
# force ; to not be seen as a tensor index

;::Symbol;
```

There remains one issue (before looking at the complete code) – How can Cadabra be informed that the connection is symmetric? Cadabra does support the `::Symmetric` and `::AntiSymmetric` properties but these apply to *all* of the indices of the attached objects. For the caee of $\Gamma^a{}_{bc}$, which is symmetric only on the lower pair of indices, Cadabra provides a more sophisticated property as follows

```
\Gamma^{a}_{b c}::TableauSymmetry(shape={2}, indices={1,2});
```

This does look a bit cryptic so a brief explanation of the syntax would be helpful. But doing so at this stage will take the discussion to far from the current objective – to compute the Riemann tensor. Thus a deeper explanation will be deferred until after the main results have been presented. Here now is the complete Cadabra code.

```
{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u#}::Indices(position=independent).

\partial{#}::PartialDerivative.
```

```
 4
 5   \Gamma^{a}_{b c}::TableauSymmetry(shape={2}, indices={1,2});
 6
 7   # force ; to not be seen as a tensor index
 8
 9   ;::Symbol;
10
11   # rules for the first two covariant derivs of V^a
12
13   deriv1 := V^{a}_{; b}      -> \partial_{b}{V^{a}}
14                               + \Gamma^{a}_{c b} V^{c}.
15
16   deriv2 := V^{a}_{; b ; c}  -> \partial_{c}{V^{a}_{; b}}
17                               + \Gamma^{a}_{d c} V^{d}_{; b}
18                               - \Gamma^{d}_{b c} V^{a}_{; d}.
19
20   substitute (deriv2,deriv1)
21
22   # commute the second covariant derivatives
23
24   Vabc := V^{a}_{; b ; c} - V^{a}_{; c ; b}.
25
26   substitute (Vabc,deriv2)
27
28   distribute     (Vabc)
29   product_rule   (Vabc)
30
31   # tidy up
32
33   sort_product   (Vabc)
34   rename_dummies (Vabc)
35   canonicalise   (Vabc)
36   sort_sum       (Vabc)
37   factor_out     (Vabc,$V^{a?}$)
```

The three rules used in the above code are reported by Cadabra as follows

$$V^a{}_{;b} \rightarrow \partial_b V^a + \Gamma^a{}_{cb} V^c \qquad /13/$$

$$V^a{}_{;b;c} \rightarrow \partial_c V^a{}_{;b} + \Gamma^a{}_{dc} V^d{}_{;b} - \Gamma^d{}_{bc} V^a{}_{;d} \qquad /16/$$

$$V^a{}_{;b;c} \rightarrow \partial_c \left(\partial_b V^a + \Gamma^a{}_{db} V^d\right) + \Gamma^a{}_{dc} \left(\partial_b V^d + \Gamma^d{}_{eb} V^e\right) - \Gamma^d{}_{bc} \left(\partial_d V^a + \Gamma^a{}_{ed} V^e\right) \qquad /20/$$

The last of these (obtained by substituting the first rule into the second) can be used to expand $V^a{}_{;b;c} - V^a{}_{;c;b}$. This leads to

$$\begin{aligned} V^a{}_{;b;c} - V^a{}_{;c;b} = {} & \partial_c \left(\partial_b V^a + \Gamma^a{}_{db} V^d\right) + \Gamma^a{}_{dc} \left(\partial_b V^d + \Gamma^d{}_{eb} V^e\right) - \Gamma^d{}_{bc} \left(\partial_d V^a + \Gamma^a{}_{ed} V^e\right) \\ & - \partial_b \left(\partial_c V^a + \Gamma^a{}_{dc} V^d\right) - \Gamma^a{}_{db} \left(\partial_c V^d + \Gamma^d{}_{ec} V^e\right) + \Gamma^d{}_{cb} \left(\partial_d V^a + \Gamma^a{}_{ed} V^e\right) \qquad /26/ \end{aligned}$$

Now the simplifications begin. First the brackets are expanded

$$\begin{aligned} V^a{}_{;b;c} - V^a{}_{;c;b} = {} & \partial_{cb} V^a + \partial_c \left(\Gamma^a{}_{db} V^d\right) + \Gamma^a{}_{dc} \partial_b V^d + \Gamma^a{}_{dc} \Gamma^d{}_{eb} V^e - \Gamma^d{}_{bc} \partial_d V^a - \Gamma^d{}_{bc} \Gamma^a{}_{ed} V^e - \partial_{bc} V^a \\ & - \partial_b \left(\Gamma^a{}_{dc} V^d\right) - \Gamma^a{}_{db} \partial_c V^d - \Gamma^a{}_{db} \Gamma^d{}_{ec} V^e + \Gamma^d{}_{cb} \partial_d V^a + \Gamma^d{}_{cb} \Gamma^a{}_{ed} V^e \qquad /28/ \end{aligned}$$

followed by the product rule

$$\begin{aligned}V^a{}_{;b;c} - V^a{}_{;c;b} = {} & \partial_{cb}V^a + \partial_c\Gamma^a{}_{db}V^d + \Gamma^a{}_{dc}\Gamma^d{}_{eb}V^e - \Gamma^d{}_{bc}\partial_d V^a - \Gamma^d{}_{bc}\Gamma^a{}_{ed}V^e - \partial_{bc}V^a - \partial_b\Gamma^a{}_{dc}V^d \\ & - \Gamma^a{}_{db}\Gamma^d{}_{ec}V^e + \Gamma^d{}_{cb}\partial_d V^a + \Gamma^d{}_{cb}\Gamma^a{}_{ed}V^e \qquad /29/\end{aligned}$$

Notice that some obvious cancelations have not been made (e.g., the $\partial^2_{bc}V^a$ terms could be cancelled). These cancellations (and other minor aesthetic improvements) will be handled by the `canonicalise` algorithm. In order to allow `canonicalise` to catch as many simplifications as possible it is common to do some basic housekeeping on the expression before calling `canonicalise`. In most cases it is sufficient to sort the products then rename the dummy indices. This leads to

$$\begin{aligned}V^a{}_{;b;c} - V^a{}_{;c;b} &= \partial_{cb}V^a + V^d\partial_c\Gamma^a{}_{db} + V^e\Gamma^a{}_{dc}\Gamma^d{}_{eb} - \Gamma^d{}_{bc}\partial_d V^a - V^e\Gamma^a{}_{ed}\Gamma^d{}_{bc} - \partial_{bc}V^a - V^d\partial_b\Gamma^a{}_{dc} \\ &\quad - V^e\Gamma^a{}_{db}\Gamma^d{}_{ec} + \Gamma^d{}_{cb}\partial_d V^a + V^e\Gamma^a{}_{ed}\Gamma^d{}_{cb} \qquad /33/ \\ &= \partial_{cb}V^a + V^d\partial_c\Gamma^a{}_{db} + V^d\Gamma^a{}_{ec}\Gamma^e{}_{db} - \Gamma^d{}_{bc}\partial_d V^a - V^d\Gamma^a{}_{de}\Gamma^e{}_{bc} - \partial_{bc}V^a - V^d\partial_b\Gamma^a{}_{dc} \\ &\quad - V^d\Gamma^a{}_{eb}\Gamma^e{}_{dc} + \Gamma^d{}_{cb}\partial_d V^a + V^d\Gamma^a{}_{de}\Gamma^e{}_{cb} \qquad /34/ \\ &= V^d\partial_c\Gamma^a{}_{bd} + V^d\Gamma^a{}_{ce}\Gamma^e{}_{bd} - V^d\partial_b\Gamma^a{}_{cd} - V^d\Gamma^a{}_{be}\Gamma^e{}_{cd} \qquad /35/\end{aligned}$$

The final pair of lines in the above code massages the Cadabra output into a familiar form

$$\begin{aligned}V^a{}_{;b;c} - V^a{}_{;c;b} &= V^d\partial_c\Gamma^a{}_{bd} - V^d\partial_b\Gamma^a{}_{cd} - V^d\Gamma^a{}_{be}\Gamma^e{}_{cd} + V^d\Gamma^a{}_{ce}\Gamma^e{}_{bd} \qquad /36/ \\ &= V^d\left(\partial_c\Gamma^a{}_{bd} - \partial_b\Gamma^a{}_{cd} - \Gamma^a{}_{be}\Gamma^e{}_{cd} + \Gamma^a{}_{ce}\Gamma^e{}_{bd}\right) \qquad /37/\end{aligned}$$

As noted above, the syntax involving the `::TableauSymmetry` does require some (limited) explanation. Cadabra uses sophisticated algorithms to handle tensor symmetries based on the Littlewood-Richardson algorithm for finding a basis of the irreducible representations of totally symmetric groups. The algorithm uses Young diagrams which consist of a set of cells arranged as series of rows which in turn are described by the `::TableauSymmetry` property. In short, the index symmetries of a tensor are encoded in these diagrams. The `shape={...}` parameter describes the shape of a Young diagram, in this case it consists of one row with two cells. The `indices={...}` parameter describes how the tensor's indices are assigned to the cells. For this purpose, the indices on the tensor are counted from left to right starting with zero. So in the above example the lower two indices b and c are counted as 1 and 2 and they are assigned to the two cells of the Young diagram. More details on using tableaux as a way to describe tensor symmetries can be found in the Cadabra manual.

3.1 A cheap hack for a symmetric connection

If Young diagrams and tableaux seem a bit too cryptic then there is a (less than ideal) alternative. One way to obtain a symmetric connection is to temporarily put $\Gamma^a{}_{bc} = G^a G_{bc}$ where $G_{bc} = G_{cb}$, ask Cadabra to make its simplifications and then return the $\Gamma^a{}_{bc}$ to the result. This is not a mathematical operation, it is just a trick to help Cadabra spot what symmetries are available. Here is a fragment of code that does the job (in the absence of any `::TableauSymmetry`)

```
...
# trick to impose zero torsion (symmetric connection)

G_{a b}::Symmetric.

substitute     (Vabc,$\Gamma^{a}_{b c} ->  G^{a} G_{b c}$)
sort_product   (Vabc)
rename_dummies (Vabc)
canonicalise   (Vabc)
substitute     (Vabc,$G^{a} G_{b c} -> \Gamma^{a}_{b c}$,repeat=True)

# tidy up and display the results
...
```

The problem with this approach is that if the pair of terms G^a and G_{bc} ever get separated (e.g., from a product rule) then it may not be possible to complete the last step of this trick, that is, to eliminate the G^a and G_{ab} in favour of $\Gamma^a{}_{bc}$. Another road to danger lies in playing this trick when products of connections are involved. For example, using this trick on $\Gamma^a{}_{bc}\Gamma^d{}_{ef} - \Gamma^d{}_{bc}\Gamma^a{}_{ef}$ would cause all terms to cancel giving zero as the result. This is clearly wrong. But if it can be shown that such problems can not arise (e.g., there are no derivatives or the equations are linear in the connection) then this method is rather easy to apply. It also provides a quick way to implement more complicated symmetries (e.g., if A_{abcde} is symmetric in the first two and last three indices put $A_{abcde} = G_{ab}G_{cde}$ with both G_{ab} and G_{abc} declared as `::Symmetric`).

Note the use of `repeat=True` in the call to `substitute` in the above code. It ensures that every product G^aG_{bc} is replaced with $\Gamma^a{}_{bc}$. This point is explored further in Exercise 3.10 below.

Exercises

3.1. Write one or more Cadabra codes to verify the following symmetries of $R^a{}_{bcd}$

$$\begin{aligned} 0 &= R^a{}_{bcd} + R^a{}_{bdc} \\ 0 &= R^a{}_{bcd} + R^a{}_{dbc} + R^a{}_{cdb} \\ 0 &= R^a{}_{bcd;e} + R^a{}_{bec;d} + R^a{}_{bde;c} \end{aligned}$$

3.2. Rewrite the code given in the above text for $R^a{}_{bcd}$ to use ∇ as the derivative operator rather than the symbol ;.

Hint: You may want to look back at Exercise 2.5.

3.3. Using

$$\begin{aligned} \partial_c g_{ab} &= \Gamma^d{}_{ac} g_{db} + \Gamma^d{}_{bc} g_{ad} \\ R^a{}_{bcd} &= \partial_c \Gamma^a{}_{bd} - \partial_d \Gamma^a{}_{bc} + \Gamma^e{}_{bd}\Gamma^a{}_{ce} - \Gamma^e{}_{bc}\Gamma^a{}_{de} \end{aligned}$$

write a Cadabra code to express $R_{abcd} = g_{ae}R^e{}_{bcd}$ in terms of $\Gamma^a{}_{bc}$, Γ_{abc} and their partial derivatives.

3.4. Use the result of the previous exercise to verify that

$$R_{abcd} = -R_{bacd}$$
$$R_{abcd} = R_{cdab}$$

Hint: Rewrite the equations in the form $Q = 0$ (for some suitable choice of Q) then use Cadabra to evaluate and simplify the left hand side.

3.5. Use Cadabra to verify that

$$(\nabla_d \nabla_c - \nabla_c \nabla_d)(A_a B_b) = B_b (\nabla_d \nabla_c - \nabla_c \nabla_d) A_a + A_a (\nabla_d \nabla_c - \nabla_c \nabla_d) B_b$$

This exercise involves little more than successive applications of the product rule. You do not need to express ∇ in terms of the connection.

The above result leads to a nice simplification. Define a new derivative operator D_{ab} by

$$D_{ab} = \nabla_b \nabla_a - \nabla_a \nabla_b$$

then the above result can be written as

$$D_{cd}(A_a B_b) = D_{cd}(A_a) B_b + A_a D_{cd}(B_b)$$

This is easy to remember – it has the form of a product rule for a typical derivative operator.

3.6. Suppose $R_{abcd} = A_a B_b C_c D_d$. Use the D operator introduced in the previous exercise to verify that

$$(\nabla_f \nabla_e - \nabla_e \nabla_f) R_{abcd} = R_{gbcd} R^g{}_{aef} + R_{agcd} R^g{}_{bef} + R_{abgd} R^g{}_{cef} + R_{abcg} R^g{}_{def}$$

You may need the following equation

$$(\nabla_c \nabla_b - \nabla_b \nabla_c) V_a = D_{bc}(V_a) = R^d{}_{abc} V_d$$

3.7. This exercise is a variation of the previous exercise – it is the full computation made without any tricks or assumptions on the form of R_{abcd}.

You will find it easier to use the symbol ; as the derivative operator rather than ∇ (as per Example 3). You will also need to create rules for both the first and second covariant derivatives of R_{abcd} and, for the final step, a rule to recover $R^a{}_{bcd}$ from terms involving the connection and its partial derivatives.

This exercise requires a lot more work than the previous exercise. Do not try to write a complete code from scratch. Start with a trivial code. Then extend that code one line at a time looking closely at Cadabra's output before choosing the next Cadabra statement.

3.8. Another standard result in differential geometry is that the Ricci tensor R_{ab} for a symmetric connection is itself symmetric. That is, given

$$\begin{aligned}0 &= R^{a}{}_{bcd} + R^{a}{}_{dbc} + R^{a}{}_{cdb}\\ 0 &= R_{abcd} + R_{bacd}\\ 0 &= R_{abcd} + R_{abdc}\end{aligned}$$

then

$$R_{ab} = R^{c}{}_{acb} = R_{ba}$$

The same result can also be obtained directly using

$$\begin{aligned}\partial_c g^{ab} &= -g^{ae} g^{bf} \partial_c g_{ef}\\ \Gamma^{a}{}_{bc} &= \frac{1}{2} g^{ad} \left(\partial_b g_{dc} + \partial_c g_{bd} - \partial_d g_{bc}\right)\\ R^{a}{}_{bcd} &= \partial_c \Gamma^{a}{}_{bd} - \partial_d \Gamma^{a}{}_{bc} + \Gamma^{e}{}_{bd}\Gamma^{a}{}_{ce} - \Gamma^{e}{}_{bc}\Gamma^{a}{}_{de}\end{aligned}$$

Use this last set of equations as a basis for a Cadabra code to verify that $R_{ab} = R_{ba}$.

3.9. Adapt your Cadabra code from the previous exercise to express R_{ab} solely in terms of g_{ab}, its first and second partial derivatives and g^{ab}. Your answer should not contain any partial derivatives of g^{ab}.

3.10. The code given in Example 3.1 included the following line

```
substitute (Vabc,$G^{a} G_{b c} -> \Gamma^{a}_{b c}$,repeat=True)
```

Is the `repeat=True` argument really necessary? Modify the source for Example 3 by removing this argument, run the new code and observe the result. You should see that the result differs from the original result. This behaviour can be understood using the following simplified code.

```
foo := A B + A B A B + A B A B A B + A B A B A B A B .
bah := A B + A B A B + A B A B A B + A B A B A B A B .

substitute (foo,$A B -> A$)
substitute (bah,$A B -> A$,repeat=True)
```

Before the two calls to `substitute` both `foo` and `bah` will equal $AB+ABAB+ABABAB+ABABABAB$ and after the two calls to `substitute` their values will be

$$\begin{aligned}\texttt{foo} &= A + AAB + AABAB + AABABAB\\ \texttt{bah} &= A + AA + AAA + AAAA\end{aligned}$$

By inspection it is easy to infer the action of `substitute` with and without the `repeat=True` argument. Without the `repeat=True` argument only the first occurrence of the target in each product term will be substituted. In contrast, when `repeat=True` argument is used Cadabra will repeat the process until all possible matches have been made.

4 Feel the Function

Since Cadabra's core language is built on Python (and implemented in C++ for efficiency) it inherits all of the functionality of Python including the use of functions. Here is a simple example of a function in Cadabra

```
def Tidy (expr):
   sort_product    (expr)
   rename_dummies (expr)
   canonicalise    (expr)
   return expr
```

This function takes a single argument expr, applies a sequence of Cadabra algorithms to expr and finally returns the updated version of expr. Since Tidy is a Python function, it must conform to all of Python's rules governing functions in particular the use of a consistent indentation in the body of the function. The function can be called using

```
foo = Tidy (bah)
```

This applies Tidy to bah and saves the result in foo. Note that this is a pure Python statement and thus the assignment is made using = rather than :=. This also explains the absence of a Cadabra statement terminator (such as ;) – it is a Python statement and thus it should conform to the Python's rules for statement termination.

As a more involved example consider the situation where you are asked to extract the cubic terms from a polynomial such as

```
Quartic := c^{a}
         + c^{a}_{b} x^b
         + c^{a}_{b c} x^b x^c
         + c^{a}_{b c d} x^b x^c x^d
         + c^{a}_{b c d e} x^b x^c x^d x^e.
```

One approach (there are others, e.g., emulating a truncated Taylor series) is to use Cadabra's ::Weight property and the keep_weight algorithm. The idea is to assign (invisible) weights to the terms of the polynomial (through the ::Weight property) and then extract terms matching a chosen weight (using the keep_weight algorithm).

Here is a Cadabra function that does the job.

```
def truncate (poly,n):

    # define the weight and give it a label
    x^{a}::Weight(label=\epsilon).

    # start with an empty expression
    ans = Ex("0")

    # loop over selected terms in the source
    for i in range (0,n+1):

       foo := @(poly).
       bah  = Ex("\epsilon = " + str(i))

```

```
15        # extract a single term
16        keep_weight (foo, bah)
17
18        # update the running sum
19        ans = ans + foo
20
21     # all done, return final answer
22     return ans
```

Though this function follows a fairly standard idiom – start with an empty sum and loop over all terms while updating the rolling sum – some elements of the syntax have not been described so far and thus a few lines of explanation are warranted.

Line 4 identifies x^a as the target to carry the weights (and is given the label `\epsilon` to distinguish it from other targets declared by other instances of `::Weight`). Cadabra now sees the polynomial as if it had been written as

```
Quartic := c^{a}
         + c^{a}_{b} x^b \eps
         + c^{a}_{b c} x^b x^c \eps^2
         + c^{a}_{b c d} x^b x^c x^d \eps^3
         + c^{a}_{b c d e} x^b x^c x^d x^e \eps^4.
```

The function `Ex` (used in lines 7 and 13) is a Cadabra function that takes a string (or zero) and returns a Cadabra expression for that string. Thus line 7 sets the rolling sum `ans` to zero while line 13 sets the target `bah` for the next term to extract from the polynomial. The syntax `foo := @(poly)` is Cadabra's way of creating a fresh copy of `poly` and saving it in `foo`. Line 16 extracts the desired term from `foo` and overwrites `foo` with the result (as per most Cadabra algorithms). The Python `for-loop` starts with `i=0` and continues for `n+1` iterations thus covering the range `i=0,1,2,...n`.

The function could be called as follows

```
Cubic = truncate
(Quartic,3)
```

with the final result exactly as expected – the leading cubic part of the original quartic polynomial.

Exercises

In each of the following exercises you can assume that each polynomial is of the form

$$p(x) = c^a + c^a{}_b x^b + c^a{}_{bc} x^b x^c + c^a{}_{bcd} x^b x^c x^d + c^a{}_{bcde} x^b x^c x^d x^e$$

where the coefficients c^a, $c^a{}_b$, $c^a{}_{bc}$ etc. may vary from one polynomial to another. The restriction to quartics is just to avoid the complexities that might otherwise arise.

4.1. Write a function that returns the first partial derivative of a polynomial, that is $\partial_b(p(x))$. For a quadratic such as

$$p(x) = c^a + c^a{}_b x^b + c^a{}_{bc} x^b x^c$$

the function should return

$$\partial_b(p(x)) = c^a{}_b + c^a{}_{bc} x^c + c^a{}_{cb} x^c$$

Hint: Begin by making substitutions like $x^a \to x^a + \delta^a$ then expand in powers of δ^a. At a later stage in your function you will need to make a second substitution $\delta^a \to 1$. This last step is not without its risks as you will discover in the following exercise.

Note. There are other (better) ways to differentiate expressions. One such method can be found following the solution for this exercise.

4.2. The solution for the previous exercise contains the following three lines

```
sort_product    (foo)
rename_dummies  (foo)
factor_out      (foo,$\delta^{a?}$)
```

Comment out those lines and then re-run the code. Look carefully at the output. Are you worried? You should be! Look at the free indices – (a, b) for the first and third terms and (a, c) on the second term. The source of this problem is the substitution `\delta^{a} -> 1`. This changes the index structure and is thus a very risky operation. It should only ever be used when it is clear that problems such as that just seen can not arise. One way to prepare an expression for rules like `\delta^{a} -> 1` is to use `sort_product`, `rename_dummies`, `canonicalise` and `factor_out` to ensure that the expression contains just one instance of δ^a.

4.3. Write a function that accepts two polynomials and returns their product truncated to a given order. The easiest solution is to multiply both polynomials, expand and then truncate at the desired order. The problem with this solution is that it requires the full product to be computed which wastes both time and memory. The better solution is to construct the product term by term, starting from 0-th order and stopping at the required order. You could start by writing a function that returns a term of a given order from a polynomial (you can use the `truncate` function from the main example as a starting point). This function could then be embedded in a loop to build a single term of the product. A further loop can be used to construct all of the required terms.

Note. When testing your function do ensure that the free indices on the two polynomials do not clash.

4.4. Here is a simple expression that is crying out for some TLC.

$$p(x) = \frac{1}{3} A_{ab} x^a x^b + \frac{1}{9} B_{ec} x^c x^e - \frac{1}{5} C_{pc} B_{dq} g^{cd} x^p x^q$$

The formatting could definitely be improved by factoring out the x^a and by clearing the fractions. Write a function with two arguments – the above polynomial and a scale factor. The scale factor should be used to clear the numerators. Your function should return the following expression

$$p(x) = \frac{1}{45} x^a x^b \left(15 A_{ab} + 5 B_{ab} - 9 B_{ca} C_{bd} g^{dc}\right)$$

4.5. This is a simple extension of the previous exercise. This time the messy polynomial is

$$\begin{aligned}p(x) = \frac{1}{7}A_e x^e - \frac{1}{3}B_f x^f \\ + \frac{1}{3}A_{ab}x^a x^b + \frac{1}{9}B_{ec}x^c x^e - \frac{1}{5}C_{pc}B_{dq}g^{cd}x^p x^q \\ + \frac{3}{7}A_{abc}x^a x^b x^c - \frac{1}{5}B_{ab}C_{cde}g^{cd}x^a x^b x^e + \frac{7}{11}B_{ab}B_{cd}C_{efg}g^{bc}g^{df}x^a x^e x^g\end{aligned}$$

This expression contains 1st, 2nd and 3rd order terms in x^a. Write a function that first extracts the 1st, 2nd and 3rd order terms then tidies each using a function based on that from the previous exercise. Finally, rebuild the expression using the three (tidy) terms. You should obtain

$$\begin{aligned}p(x) = \frac{1}{21}x^a\left(3A_a - 7B_a\right) \\ + \frac{1}{45}x^a x^b\left(15A_{ab} + 5B_{ab} - 9B_{ca}C_{bd}g^{dc}\right) \\ + \frac{1}{385}x^a x^b x^c\left(165A_{abc} - 77B_{ab}C_{dec}g^{de} + 245B_{ad}B_{ef}C_{bgc}g^{de}g^{fg}\right)\end{aligned}$$

4.6. As noted in an earlier Exercise 1.6, successive instances of `SortOrder` might not produce the desired result (e.g., using `{A,B}::SortOrder` as an attempt to undo a previous `{B,A}::SortOrder` will fail). How can such problems be avoided? If the expression that needs to be sorted is composed solely of items with names like `AAA01`, `AAA02`, `AAA03` etc. then the sorting can be done using Cadabra's default sort order (i.e., no need to declare `SortOrder`).

Write a function that uses the `substitute` algorithm to replace targeted objects with names like `AAA01`, `AAA02`, `AAA03` etc. Use this as a basis to sort the expression. Then undo the substitutions and return the now sorted expression.

Test your function by sorting the following expression to place all of the x^a to the left of all other terms

```
expr := g_{a b} x^{a} x^{b} + \Gamma_{a b c} x^{a} x^{b} x^{c}
```

The value of this approach is that it allows you to create bespoke sort functions that will work as intended every time. The coding is certainly more tedious than using `::SortOrder` though the certainty of the result probably justifies the effort.

4.7. Since Cadabra's functions like `sort_product`, `canoniclaise` etc. can alter their argument in place it is possible to write functions like

```
def tidy (obj):
    sort_product   (obj)
    rename_dummies (obj)
    canonicalise   (obj)

foo := C^{f} B^{a} A_{f a}.
tidy (foo)
```

Notice the absence of a line like `return obj`. This function will work as expected but it is not standard Python practice. However, a function like

```
def tidy (obj):
    bah := @(obj)
    sort_product   (bah)
    rename_dummies (bah)
    canonicalise   (bah)
    obj := @(bah)

foo := C^{f} B^{a} A_{f a}.
tidy (foo)
```

will *not* return the correct result. Verify these claims by running each of the above codes and observing the result for `foo`. A good working practice is to always use a Python `return` statement to return the final result of the function.

---

5 Stay focused

When massaging an expression towards a desired form it is often the case that some terms in the expression need special attention while others can be left as they stand. One way to implement this workflow in Cadabra is to manually pull apart the expression then allow Cadabra to do its magic on the separate pieces. This is not ideal and it would be better if the expression could be left intact while restricting Cadabra's actions to targeted parts of the expression. Cadabra provides two algorithms `zoom` and `unzoom` designed to focus Cadabra's attention to specific targets in an expression.

As an example, consider the task of replacing the second `v^{a}` in the following expression with `w^{a}`

```
expr := A_{a} v^{a} + B_{a} v^{a} + C_{a} v^{a};
```

Using `substitute (expr,$v^{a}->w^{a}$)` would replace *each* instance of `v^{a}` with `w^{a}`. Thus some further information must be given to Cadabra to restrict its attention to just the middle term – this is where the `zoom` and `unzoom` algorithms enter the scene. Here is a short Cadabra fragment that uses `zoom` and `unzoom` to do the job properly.

```
1  expr := A_{a} v^{a} + B_{a} v^{a} + C_{a} v^{a};
2  zoom       (expr, $B_{a} Q??$)
3  substitute (expr, $v^{a} -> w^{a}$);
4  unzoom     (expr)
```

The corresponding output is as follows.

$$\begin{aligned} A_a v^a + B_a v^a + C_a v^a &= \ldots + B_a v^a + \ldots && /2/ \\ &= \ldots + B_a w^a + \ldots && /3/ \\ &= A_a v^a + B_a w^a + C_a v^a && /4/ \end{aligned}$$

The `zoom` algorithm is designed to zoom in on selected parts of an expression. When `zoom` is in play Cadabra will use ellipses ... to denote those parts of the expression currently hidden from view. This can be seen in lines /2/ and /3/ of the above output. The original view is recovered with the call to `unzoom` in line 4. Note that for the duration of a `zoom/unzoom` pair, Cadabra retains the full expression even though it only displays the parts selected by `zoom`.

A close look at the call to `zoom` in line 2 above reveals that `zoom` takes two arguments, the first is the expression that will be `zoom`'ed and the second is a pattern that describes the target. In this case the pattern is `B_{a}Q??`. The first part of this pattern `B_{a}` is easy to understand while the second part `Q??` is suggestive of a pattern matching rule. This is the second of Cadabra's pattern matching rules[a] – the first pattern, such as `A?_{a}`, matches any tensor with a single downstairs index, the second pattern, such as `Q??`, matches an arbitrary expression composed of sums and products of arbitrary tensors. Thus `Q??` will match each of the following expressions A^a, $A^a B_a$, $V_a + W_{abc}P^{bc}$. The pattern used in the above example was `B_{a} Q??` and thus will match only the middle term of the original expression. The upshot is that the call to `substitute` will only alter the middle term. This can be seen clearly in lines /2/ and /3/ of the above output. The final line of the output is the result of the call to `unzoom` in line 4. This shows that first and third terms where indeed left untouched by `substitute`.

[a]Cadabra also supports conditional and regular expression patterns. See https://cadabra.science/notebooks/ref_patterns.html for more details.

Note that the choice of `Q` in the pattern `Q??` is not ordained by Cadabra – any symbol could be used. Note also that these double question mark patterns can be used in substitution rules. For example, the rule `A_{a} W?? -> B_{a} W??` would replace `A_{a}` with `B_{a}` in any expression that begins with `A_{a}`.

Finally note that nesting of calls to `zoom` and `unzoom` is allowed and this can be used for greater control in selecting targets within an expression.

5.1 Tags

Suppose that V_{ab} is an anti-symmetric tensor, i.e., $V_{ab} = -V_{ba}$. Then it is clear that an expression like $2V_{ab} - 3V_{ba}$ can be reduced to just $5V_{ab}$. This reduction could easily be implemented in Cadabra using something like the following

```
V_{a b}::AntiSymmetric.
expr := 2 V_{a b} - 3 V_{b a}.
canonicalise (expr)
```

Now suppose that you wished to achieve the same result but *without* assigning the `AntiSymmetric` property to `V_{a b}`. Clearly the call to `canoncialise` will no longer swap the indices on the second term and thus the expression will remain in its original form. The challenge is to persuade Cadabra to swap the indices on the second term. This suggests a substitution like the following

```
expr := 2 V_{a b} - 3 V_{b a}.
substitute (expr, $3 V_{b a} -> - 3 V_{b a})
```

Alas, this too will fail as Cadabra will report a runtime error – numerical factors on the left of a rule, such as 3 in the above code, are not allowed. A similar problem arises when trying to use `zoom` to shift the focus. Thus the following code

```
expr := 2 V_{a b} - 3 V_{b a}.
zoom       (expr, $3 V_{b a})
substitute (expr, $V_{b a} -> - V_{b a})
```

will produce a similar runtime error.

One solution to this problem is to modify the expression by adding unique tags to each term. These tags can then be used as the targets for `zoom`. The tagged expression can then be manipulated to achieve the desired result after which the tags are removed. For the current example, reducing $2V_{ab} - 3V_{ba}$ to $5V_{ab}$, this is certainly a case of using a jet plane to cross a street but the general method is applicable to much more challenging problems (as shown in Example 12 in part 3).

The process of adding and clearing tags can be achieved with calls to the following pair of functions

```
def add_tags (obj,tag):
   n = 0
   ans = Ex('0')
   for i in obj.top().terms():
      foo = obj[i]
      bah = Ex(tag+'_{'+str(n)+'}')
```

```
 7          ans := @(ans) + @(bah) @(foo).
 8          n = n + 1
 9       return ans
10
11    def clear_tags (obj,tag):
12       ans := @(obj).
13       foo  = Ex(tag+'_{a?} -> 1')
14       substitute (ans,foo)
15       return ans
```

The operation of each function involves a simple mix of Python and Cadabra constructs. Both functions take two arguments, the first is the expression to be tagged and the second is a string that describes the base of the tag. The tag base, for example μ, is used to generate a sequence of tags such as $\mu_0, \mu_1, \mu_2, \ldots$. The `add_tags` function uses a `for-loop` to select each term in the expression (line 5), multiplies that term by the tag and then updates a rolling sum (line 7). The `clear_tags` function does its job by simply replacing all tags with the number 1.

Here is a short code fragment that demonstrates how these functions can be used to reduce $2V_{ab} - 3V_{ba}$ to $5V_{ab}$.

```
1    expr := 2 V_{p q} - 3 V_{q p}.
2
3    expr = add_tags (expr,'\\mu')
4
5    zoom       (expr, $\mu_{1} Q??$)
6    substitute (expr, $V_{a b} -> - V_{b a}$)
7    unzoom     (expr)
8
9    expr = clear_tags (expr,'\\mu')
```

The corresponding output is as follows.

$$\begin{aligned}
2V_{pq} - 3V_{qp} &= 2\mu_0 V_{pq} - 3\mu_1 V_{qp} && /3/\\
&= \ldots - 3\mu_1 V_{qp} && /5/\\
&= \ldots + 3\mu_1 V_{pq} && /6/\\
&= 2\mu_0 V_{pq} + 3\mu_1 V_{pq} && /7/\\
&= 5V_{pq} && /9/
\end{aligned}$$

The main objection to this method is that it requires explicit knowledge of the left to right order of the terms in an expression. Consider for example a working code that uses the above functions and that at some point targets the μ_7 term. If some changes are made to the code then that term may no longer be matched to μ_7. The re-ordering of the terms might now find the target term matched with μ_4. This would require corresponding changes to the calls to `zoom`. It is also possible that this same problem could arise, not through any change of the users code, but by changes made by the Cadabra team to the internal workings of its own functions. The bottom lines is that the user must take care when using these functions – careful scrutiny of the output should be standard practice!

Exercises

5.1. Verify that the following substitution rule

```
expr := A_{a} (P^{b}+Q^{b}) + C_{a} V^{b}.
swap := A_{a} B?? + C_{a} D?? -> A_{a} D?? + C_{a} B??
```

can be used to swap the expressions attached to `A_{a}` and `C_{a}`.

5.2. Verify the claim that Cadabra will report a runtime error when attempting the following substitution

```
expr := 2 V_{a b} - 3 V_{b a}.
substitute (expr, $3 V_{b a} -> - 3 V_{a b})
```

5.3. Use a suitable substitution pattern to delete the second term in the following polynomial

$$p(x) = A_{ab}B^{ab} + A_{ab}A_{cd}B^{ab}B^{cd} - C_{ab}B^{ab}$$

Do not use a tagged expression – that approach will be left for the next exercise.

5.4. Repeat the previous exercise but this time making use of the `add_tags` and `clear_tags` functions.

Hint: A simple way to delete a term is to multiply it by zero.

5.5. A common method of introducing a Riemann tensor into a computation is to make use of the simple commutation rule for covariant derivatives, namely

$$V^{a}{}_{;b;c} = V^{a}{}_{;c;b} - R^{a}{}_{dbc}V^{d}$$

Use this equation as the basis of a Cadabra code to simplify the expression $V^{a}{}_{;b;c} - V^{a}{}_{;c;b}$ to the expected result, namely $-R^{a}{}_{dbc}V^{d}$.

Hint: You will need to work with a tagged expression.

6 Full disclosure

Previous examples in this tutorial have shown that Cadabra is no slouch when it comes to complex tensor algebra. The purpose of this example is to show that it is also a dab hand at component computations, that is, given a set of coordinates, compute the components of a tensor in those coordinates.

Here is a rather simple first example. Given the components of V_a, compute the components of a new tensor dV defined by $dV_{ab} = \partial_b V_a - \partial_a V_b$. This computation entails a few basic steps – choose a set of coordinates, express the components of V in those coordinates then evaluate dV. The information required by Cadabra is much the same as just described – a set of coordinates, the components of V_a and a way to compute dV_{ab} from V_a. Here is a short Cadabra code that does the job.

```
1   {\theta, \varphi}::Coordinate.
2   {a,b,c,d,e,f,g,h#}::Indices(values={\theta, \varphi}, position=independent).
3
4   \partial{#}::PartialDerivative.
5
6   V  := { V_{\theta} = \varphi, V_{\varphi} = \sin(\theta) }.
7   dV := \partial_{b}{V_{a}} - \partial_{a}{V_{b}}.
8
9   evaluate (dV, V)
```

The first line declares a pair of symbols for the coordinates while the second line attaches those coordinates to the indices. Note the use of Greek letters to denote the coordinates in contrast to the Latin characters used for the tensor indices. This is an aesthetic choice commonly used in research articles in General Relativity to make clear the distinction between an abstract expression for a tensor and its components in a given frame (known as the Penrose abstract index notation). Note also that only two coordinates were declared in the first line – this implies that the tensors live in a two dimensional space. The components of V_a are described in line /6/ as an explicit list of values. Each entry in the list is of the form `foo = bah` where `foo` is one of the components of a tensor (in this case V_a) while `bah` is a scalar expression (i.e., an expression that does not contain any free indices, there are other restrictions as noted below). Line /7/ informs Cadabra how to compute dV_{ab} from V_a while the final line completes the job – it evaluates each of the components of dV_{ab}. The output from the above code is

$$V_a = [V_\theta = \varphi,\ V_\varphi = \sin\theta] \qquad /6/$$

$$\partial_b V_a - \partial_a V_b = \Box_{ab} \begin{cases} \Box_{\varphi\theta} = \cos\theta - 1 \\ \Box_{\theta\varphi} = -\cos\theta + 1 \end{cases} \qquad /7/$$

The format of the output shown in line /7/ is typical of Cadabra's output for a call to `evaluate`. It displays the non-zero components as a table using a $\Box$ as a place holder for the underlying tensor. Note that the expression on the far left of line /7/ has been added here to aid in reading the output – this term was not generated by Cadabra but was included by hand. Cadabra's output begins with $\Box_{ab}$ in that same line of output. The ab subscripts on $\Box_{ab}$ are matched to the indices of the source on the left and the components on the right. Thus with $a = \theta$ and $b = \varphi$ the above output reads as

$$\partial_\varphi V_\theta - \partial_\theta V_\varphi = \Box_{\theta\varphi} = -\cos\theta + 1$$

Cadabra does place some restrictions on the scalar expressions that can be used when describing a component of a tensor (given as bah in the above paragraph). It is easier to show by example what Cadabra will or will not accept for a scalar expression rather than spelling out Cadabra's rules in detail. Here are set of definitions for V_a that are allowed by Cadabra.

```
V := { V_{\theta} = \varphi, V_{\varphi} = \sin(\theta) }.
V := { V_{\theta} = \varphi, V_{\varphi} = \partial_{\theta}{\sin(\theta)} }.
V := { V_{\theta} = f(\theta,\varphi), V_{\varphi} = g(\theta,\varphi)}.
```

In contrast, each of the following definitions will be rejected by Cadabra.

```
V := { V_{a} = W_{a} }.                              # don't use free indices
V := { V_{\theta} = \theta, V_{\varphi} = V_{\theta} }. # don't use sub/super-scripts
V := { V_{\theta} = \varphi^2, V_{\varphi} = \theta }.  # use ** for powers
```

One other notable exception is that Cadabra does not (yet) support the use of derivative operators on the left hand side of the component rules. The following code fragment will raise a Cadabra run time error.

```
\partial{#}::PartialDerivative.
V_{a}::Depends(\theta,\varphi,\partial{#}).
V := { \partial{\theta}{V_{\varphi}} = \cos(\theta) }.  # partial derivatives not supported
```

A workaround for problems like this is given later in Exercise 6.10.

6.1 Ricci curvature of a 2-sphere

This approach can be easily extended to a somewhat more realistic example – computing the Ricci tensor for a 2-sphere. The starting point in this case is the metric of a 2-sphere, which in polar coordinates (θ, φ) can be written as

$$ds^2 = r^2 \left(d\theta^2 + \sin^2\theta d\varphi^2\right)$$

The metric components are described in Cadabra by a list of non-zero components

```
gab := { g_{\theta\theta}   = r**2,
         g_{\varphi\varphi} = r**2 \sin(\theta)**2 }.
```

Since the Ricci tensor also depends on the inverse metric g^{ab} the g^{ab} must also be known to Cadabra before computing the Ricci tensor. In this simple example it is easy compute the inverse by hand and then provide a list such as

```
iab := { g^{\theta\theta}   = 1/r**2,
         g^{\varphi\varphi} = 1/(r**2 \sin(\theta)**2) }.
```

A second method is to use Cadabra's `complete` algorithm (see the source code of this example for a modified version that uses `complete`).

Note that the lists for `gab` and `iab` contain *all* of their non-zero components. These are just lists of simple expressions and Cadabra knows nothing about any symmetries that might be associated with these lists. In our case the underlying tensors are symmetric so it is essential that *all* non-zero components be included in the list including those that could be inferred from the symmetries. Thus for a metric of the form $ds^2 = du^2 + 2uvdudv + dv^2$ the components must be specified using

```
gab := { g_{u u} = 1,  g_{u v} = uv,
         g_{v u} = uv, g_{v v} = 1 }.
```

Here is the complete code that does the job.

```
{\theta, \varphi}::Coordinate.
{a,b,c,d,e,f,g,h#}::Indices(values={\theta, \varphi}, position=independent).

\partial{#}::PartialDerivative.

Gamma := \Gamma^{a}_{f g} -> 1/2 g^{a b} (   \partial_{g}{g_{b f}}
                                            + \partial_{f}{g_{b g}}
                                            - \partial_{b}{g_{f g}} ).

Rabcd := R^{d}_{e f g} ->   \partial_{f}{\Gamma^{d}_{e g}}
                          - \partial_{g}{\Gamma^{d}_{e f}}
                          + \Gamma^{d}_{b f} \Gamma^{b}_{e g}
                          - \Gamma^{d}_{b g} \Gamma^{b}_{e f}.

Rab := R_{a b} -> R^{c}_{a c b}.

gab := { g_{\theta\theta}   = r**2,
         g_{\varphi\varphi} = r**2 \sin(\theta)**2 }.

iab := { g^{\theta\theta}   = 1/r**2,
         g^{\varphi\varphi} = 1/(r**2 \sin(\theta)**2) }.

substitute (Rabcd, Gamma)
substitute (Rab, Rabcd)

evaluate   (Gamma, gab+iab, rhsonly=True)
evaluate   (Rabcd, gab+iab, rhsonly=True)
evaluate   (Rab,   gab+iab, rhsonly=True)
```

There are two minor aspects of the above code that should be noted. First, each call to `evaluate` acts on a Cadabra rule and thus the argument `rhsonly=True` is used to restrict the action of `evaluate` to just the right hand side of the rule. Second, the construction `gab+iab` results in a single list built from `gab` and `iab` (this is standard Python syntax).

The output from the above code is as follows

$$\left[g_{\theta\theta} = r^{2},\ g_{\varphi\varphi} = r^{2}(\sin\theta)^{2}\right] \qquad /17/$$

$$\left[g^{\theta\theta} = r^{-2},\ g^{\varphi\varphi} = \left(r^{2}(\sin\theta)^{2}\right)^{-1}\right] \qquad /20/$$

$$\Gamma^{a}{}_{fg} \rightarrow \Box_{fg}{}^{a} \begin{cases} \Box_{\varphi\theta}{}^{\varphi} = (\tan\theta)^{-1} \\ \Box_{\theta\varphi}{}^{\varphi} = (\tan\theta)^{-1} \\ \Box_{\varphi\varphi}{}^{\theta} = -\dfrac{1}{2}\sin(2\theta) \end{cases} \qquad /26/$$

$$R^{d}{}_{efg} \rightarrow \Box_{eg}{}^{d}{}_{f} \begin{cases} \Box_{\varphi\varphi}{}^{\theta}{}_{\theta} = (\sin\theta)^{2} \\ \Box_{\theta\varphi}{}^{\varphi}{}_{\theta} = -1 \\ \Box_{\varphi\theta}{}^{\theta}{}_{\varphi} = -(\sin\theta)^{2} \\ \Box_{\theta\theta}{}^{\varphi}{}_{\varphi} = 1 \end{cases} \qquad /27/$$

$$R_{ab} \rightarrow \Box_{ab} \begin{cases} \Box_{\varphi\varphi} = (\sin\theta)^{2} \\ \Box_{\theta\theta} = 1 \end{cases} \qquad /28/$$

The above results are mostly self-explanatory. However, the notation used in displaying the Riemann components does require a brief explanation. Note that the order of the indices on the left and right hand sides do not match. Despite this fact, the indices do maintain a strict one-to-one correspondence. For example, the component $R^{\theta}{}_{\varphi\theta\varphi}$ has indices $d = \theta, e = \varphi, f = \theta$ and $g = \varphi$ which, on the right hand side, corresponds to $\Box_{\varphi\varphi}{}^{\theta}{}_{\theta}$. Thus $R^{\theta}{}_{\varphi\theta\varphi} = \Box_{\varphi\varphi}{}^{\theta}{}_{\theta} = \sin^2\theta$.

There is one small variation on the above code that is worth noting. Consider the following small addition to the code

```
Riem := R^{d}_{e f g}.
substitute (Riem, Rabcd)
evaluate   (Riem, gab+iab)
```

The object `Riem` is no longer a substitution rule but rather a simple Cadabra expression. Thus there is no need in this case for the `rhsonly=True` argument in the call to `evaluate`. The output from the above code is

$$\Box_{eg}{}^{d}{}_{f} \begin{cases} \Box_{\varphi\varphi}{}^{\theta}{}_{\theta} = (\sin\theta)^{2} \\ \Box_{\theta\varphi}{}^{\varphi}{}_{\theta} = -1 \\ \Box_{\varphi\theta}{}^{\theta}{}_{\varphi} = -(\sin\theta)^{2} \\ \Box_{\theta\theta}{}^{\varphi}{}_{\varphi} = 1 \end{cases}$$

which no longer contains the informative left hand side, that is, $R^{d}{}_{efg} \rightarrow$. So when writing code it may be useful to apply the `evaluate` algorithm to a rule (if convenient) rather than a simple expression. The advantage in doing so is that the left hand side of the rule retains a useful reminder of how the indices map to the components on the right hand side.

6.2 Selecting components

Calls to `evaluate` will return a Cadabra object that contains all of the non-zero components. This raises a simple question – How can individual components of a tensor be found? There are two simple answers, the first takes a sideways step by using `evaluate` to compute a scalar function that matches the desired component while the second is a direct method that extracts the target component from the full list of components.

So, how can the $\varphi\varphi$ component of the metric tensor g_{ab} of the 2-sphere be found? The direct approach is to compute the projection of the tensor onto the appropriate basis vectors, in this case g_{ab} on $e_\varphi = \partial_\varphi$. That is, compute $g_{ab}e^a_\varphi e^b_\varphi$. In the following code Cadabra is asked to `evalaute` the scalar function $g_{ab}e^a_\varphi e^b_\varphi$.

```
{\theta, \varphi}::Coordinate.
{a,b,c,d,e,f,g,h#}::Indices(values={\theta, \varphi}, position=independent).

theta{#}::LaTeXForm{"\theta"}.
varphi{#}::LaTeXForm{"\varphi"}.

gab := { g_{\theta \theta}   = r**2,
         g_{\varphi \varphi} = r**2 \sin(\theta)**2 }.

# obtain components by contracting with basis vectors

basis := {theta^{\theta} = 1, varphi^{\varphi} = 1}.

compt := g_{a b} varphi^{a} varphi^{b}.

evaluate (compt,gab+basis)
```

The output from the above code is

$$\begin{aligned} g_{\varphi\varphi} &= g_{ab}\varphi^{a}\varphi^{b} && /14/ \\ &= r^{2}(\sin\theta)^{2} && /16/ \end{aligned}$$

Now consider the second approach where the desired component is to be extracted after a call to `evaluate`. A starting point for the code might be

```
{\theta, \varphi}::Coordinate.
{a,b,c,d,e,f,g,h#}::Indices(values={\theta, \varphi}, position=independent).

theta{#}::LaTeXForm{"\theta"}.
varphi{#}::LaTeXForm{"\varphi"}.

gab := { g_{\theta \theta}   = r**2,
         g_{\varphi \varphi} = r**2 \sin(\theta)**2 }.

metric := g_{a b}.

evaluate (metric,gab)
```

What is missing from this code is the crucial lines that extracts the target from `metric`.

Since Cadabra is intimately coupled to Python it is not surprising that it inherits Python's array-indexing notation. Thus if `foo` is a Cadabra object then elements of that object can be accessed using notation like `foo[0]`, `foo[1]`, `foo[2]` etc. Deeper probing into `foo` is allowed using notation like `foo[2][1]`, `foo[2][1][3]` etc. This is standard Python notation for Python arrays. However, Cadabra has its own way of storing the elements of its objects. For the metric of the 2-sphere Cadabra needs to store not just the components of g_{ab} but also the index pairs for each component, the free indices and information describing the position of each index (up or down). For this discussion, the bulk of that data can be summarised as a list like

```
metric = {a,b,{{\theta, \theta} = (r)**2,
               {\varphi, \varphi} = (r)**2 (\sin(\theta))**2}}
```

Note that the line break is just for presentation (it has no syntactic meaning). Note also that this format is just for discussion – it is neither correct Python notation nor is it a literal copy of Cadabra's own representation. The list (after `metric =`) begins with the free indices, `a,b`. It then contains a sub-list of terms of the form `foo=bah` where `foo` records an index pair and where `bah` is the corresponding component. Note that the sub-list of components counts as one item in the main list. Finally, each entry like `foo=bah` is treated as a list with `foo` and `bah` taken as elements 0 and 1 respectively. This is all that needs be known in order to access the elements of this list. Thus

```
metric[0] = a
metric[1] = b
metric[2] = {{\theta, \theta} = (r)**2,
             {\varphi, \varphi} = (r)**2 (\sin(\theta))**2}
metric[2][1] = {\varphi, \varphi} = (r)**2 (\sin(\theta))**2
metric[2][1][0] = {\varphi, \varphi}
metric[2][1][1] = (r)**2 (\sin(\theta))**2
```

and so on. Thus in this case the one extra line required in the above code to extract the target component is simply

```
compnt = metric[2][1][1]
```

The one important caveat with this method is that it may require a spot of detective work to find the target item. The lists that Cadabra uses are not ordered lists as there is no unique way to map the multi-dimensional set of indices to a single dimension (the position in the list). Thus for other tensors, say `Aab := A_{a b}`, the $\varphi\varphi$ component might actually be found at `Aab[2][0][1]`. So tread carefully when using this method. If in doubt, either repeat the above exercise (of stepping through the indices) or revert to the first method (projection onto the basis vectors).

6.3 Components in pure Python/SymPy

It is quite likely that one of the reasons for extracting one or more components of tensor is that some numerical values are sought (e.g., for plotting or for use in a separate numerical simulation).

How can a Cadabra expression be evaluated numerically? This clearly sounds like a job for Python/SymPy (or NumPy). But first the Cadabra expression, which may contain LaTeX markup, needs to be reformatted for use by Python. This is rather easy – just apply the `._sympy_()` method to convert the Cadabra expression to a SymPy expression. Note that the `._sympy()` method is the counterpart to Cadabra's `Ex` function (which converts strings to Cadabra expressions).

Thus a Python expression for the $\varphi\varphi$ component of the 2-metric could be created using

```
r, theta, varphi = symbols('r theta varphi')
gphiphi = compt._sympy_()
```

where `compt` is the result obtained above using the projection method. The first line is needed only when `gphiphi` will be subsequently processed by SymPy operations. Note that the usual Python line

```
from sympy import *
```

is not needed as this is always included by Cadabra as part of its initialisation.

You can verify that the before and after expressions have the expected types by using the following code fragment

```
print ('type compt   = ' + str(type(compt)))
print ('type gphiphi = ' + str(type(gphiphi)))
```

This produces the following output

```
type compt   = <class 'cadabra2.Ex'>
type gphiphi = <class 'sympy.core.mul.Mul'>
```

Exercises

6.1. Modify the original example by replacing line 7 with

```
7   dV := dV_{a b} -> \partial_{b}{V_{a}} - \partial_{a}{V_{b}}.
```

Observe the output then repeat using

```
7   dV := dV_{a b} -> \partial_{b}{V_{a}} - \partial_{a}{V_{b}}.
8   evaluate (dV, V, rhsonly=True)
```

6.2. Modify the original example by replacing lines 6 and 7 with

```
6   V  := { V_{\theta} = f(\theta,\varphi), V_{\varphi} = g(\theta,\varphi) }.
7   dV := \partial_{b}{V_{a}} + \partial_{a}{V_{b}}.
```

Run the new code and observe the output. Not much to say here other than to admire your handiwork.

6.3. When processing a statement like `evaluate(foo,bah)` Cadabra will use `bah` as a pool of expressions to fulfil any requests while evaluating each component of `bah`. What happens if the pool does not contain the requested component? If the pool contains some but not all entries for a tensor then the remaining entries are taken to be zero. Now run a code built on this fragment

```
bah  := {V_{\theta} = \varphi, V_{\varphi} = \sin(\theta)}.
foo  := U_{a} V_{b}.
evaluate (foo, bah)
```

The output should show that Cadabra has assumed that all entries of U_a are non-zero despite there being no entries for U_a in the pool `bah`.

6.4. Extend the sample code for the 2-sphere to also compute the scalar curvature. The result should be $2/r^2$ (as expected).

6.5. Look back at the black magic that revealed how to access the $\varphi\varphi$ component of the metric tensor. That showed that `metric[2][1][1]` gave the desired component, namely $g_{\varphi\varphi}$. But when `evaluate` is applied to a *rule* a different indexing is required. Use carefully chosen `print(metric[...])` commands to find the new black magic for such a case.

Note. You will need to modify the code slightly so that `evaluate` acts on a rule rather than a simple expression.

Hint: Start with something simple like `print(metric[0])` and `print(metric[1])`. Observe the output and then refine the choice of indices to reach the target.

6.6. Repeat the previous exercise but this time extract the $R_{\varphi\varphi}$ component of the Ricci tensor. Look carefully at the indices. You may think that an index selection like `[2][1][1]` will apply for both g_{ab} and R_{ab}, after all, they do share the same index structure so surely Cadabra will store the components in exactly the same order. But do remember that the lists that Cadabra uses to store the components are not sorted – thus the entries could appear in any order. Caveat emptor.

6.7. Verify that the Schwarzschild metric in isotropic coordinates

$$ds^2 = -\left(\frac{2r-m}{2r+m}\right)^2 dt^2 + \left(1+\frac{m}{2r}\right)^4 \left(dr^2 + r^2\left(d\theta^2 + \sin^2\theta\, d\phi^2\right)\right)$$

is a solution of the vacuum Einstein equations $0 = R_{ab}$.

6.8. Compute the Ricci tensor R_{ab} for the Kasner metric

$$ds^2 = -dt^2 + t^{2p_1}dx^2 + t^{2p_2}dy^2 + t^{2p_3}dz^2$$

Hence verify that the Ricci tensor vanishes provided

$$p_1 + p_2 + p_3 = p_1^2 + p_2^2 + p_3^2 = 1$$

6.9. Consider the Schwarzschild metric in Schwarzschild coordinates

$$ds^2 = -f(r)dt^2 + \frac{1}{f(r)}dr^2 + r^2\left(d\theta^2 + \sin^2\theta\, d\phi^2\right)$$

where $f(r) = (1-2m/r)$. Show that each of the following vectors

$$\begin{aligned}
(i) &\quad \xi = \xi^a\partial_a = \partial_t \\
(ii) &\quad \xi = \xi^a\partial_a = \partial_\varphi \\
(iii) &\quad \xi = \xi^a\partial_a = \sin\varphi\,\partial_\theta + \cot\theta\cos\varphi\,\partial_\varphi \\
(iv) &\quad \xi = \xi^a\partial_a = \cos\varphi\,\partial_\theta - \cot\theta\sin\varphi\,\partial_\varphi
\end{aligned}$$

is a solution of Killing's equation

$$0 = \xi_{a;b} + \xi_{b;a}$$

Hint: You will need to provide two lists of components, one for g_{ab} and one for ξ^a.
Note. To avoid a runtime error you will need to write $\cot\theta$ as $\cos\theta/\sin\theta$.

6.10. At present Cadabra does not support component rules that include derivative operators in the targets of the component definitions. Thus code like the following will raise a run time error.

```
{\theta, \varphi}::Coordinate.
{a,b,c,d,e,f,g,h#}::Indices(values={\theta, \varphi}, position=independent).

\partial{#}::PartialDerivative.

V_{a}::Depends(\theta,\varphi,\partial{#}).

dVrule := { \partial_{\theta}{V_{\varphi}} = \sin(\theta),
            \partial_{\varphi}{V_{\theta}} = \cos(\theta)}.
dV := \partial_{b}{V_{a}} - \partial_{a}{V_{b}}.

evaluate (dV, dVrule)
```

Though the intention is clear, Cadabra (at present) does not allow the rule `dVrule` to be used in the call to `evaluate`. One solution to this impasse is to hide the derivative from `evaluate` by making a substitution `\partial_{a}{V_{b}} -> dV_{a b}` then applying `evaluate` to `dV_{a b}`. Test this idea by modifying the above code to include this hack.

7 Escape to C

An increasingly popular approach in computational physics is to harness the power of programs like Mathematica and Maple to convert differential equations into computational procedures written in a language like C or Fortran (see [10] and [11]). Cadabra can take this one step further by first processing the tensor equations before handing the results over to Mathematica, Maple or even Cadabra's own internal version of SymPy. This opens up the possibility of using Cadabra, from beginning to end, by starting with a complex tensor equation, such as Einstein's equations, and then doing all the work required to produce a stand alone C program.

All but the last stage of this workflow can be easily handled using techniques described in the previous examples. The final stage of this workflow, where the C-code is created, can be easily implemented using the Codegen package from Python/Sympy. The basic idea is to iterate over a list of expressions, passing each expression to Codegen and then saving the results to a file. Here is a short Python code that writes raw C-code for a single tensor[b].

```
def write_code (obj,name,filename,rank):

   import os

   from sympy.printing.ccode import C99CodePrinter as printer
   from sympy.printing.codeprinter import Assignment

   idx=[]  # indices in the form [{x, x}, {x, y} ...]
   lst=[]  # corresponding terms [termxx, termxy, ...]

   for i in range( len(obj[rank]) ):                  # rank = num. of free indices
       idx.append( str(obj[rank][i][0]._sympy_()) )  # indices for this term
       lst.append( str(obj[rank][i][1]._sympy_()) )  # the matching term

   mat = sympy.Matrix([lst])                          # row vector of terms
   sub_exprs, simplified_rhs = sympy.cse(mat)         # optimise code

   with open(os.getcwd() + '/' + filename, 'w') as out:

      for lhs, rhs in sub_exprs:
         out.write(printer().doprint(Assignment(lhs, rhs))+'\n')

      for index, rhs in enumerate (simplified_rhs[0]):
         lhs = sympy.Symbol(name+' '+(idx[index]).replace(', ',']['))
         out.write(printer().doprint(Assignment(lhs, rhs))+'\n')
```

The function `write_code` takes four arguments. The first, `obj`, is a list of components of the tensor created in a prior call to `evaluate`. The second, `name`, is a string representing the C-array name. Entries in this array will be the C-code for the corresponding tensor component and its indices will match exactly those of the tensor represented by `obj`. The third argument is simply the filename while the final argument, `rank`, equals the number of free indices on the tensor.

The function also applies basic optimisation of the C-code by looking for common subexpressions and writes these (as assignments to variables like `x0`, `x1`, `x2` ...) to the file ahead of the tensor components.

[b]An extended vesrion of the function, suitable for use with tensors and scalars, can be found in `support/python/writecode.py`

The argument obj is assumed to be the result of a call to evaluate. As noted in Example 6 (on the 2-sphere) there are two ways to call evaluate, the first uses a simple expression as in

```
Rab := R_{a b};
evaluate (Rab, ...)
```

while the second uses a substitution rule as in

```
Rab := R_{a b} -> R^{c}_{a c b};
evaluate (Rab, ...)
```

The function write_code is designed for the first case and expects a call like write_code(Rab,...). However, if the components were created using the second method, then the correct call would be write_code(Rab[1],...). Using Rab[1] steps over the leading Rab -> part of Rab.

The connection, Riemann and Ricci components for the 2-sphere (using the Cadabra code from Example 6) could be converted to C using

```
write_code (Gamma[1], 'myGamma', 'example-07-gamma.c', 3)
write_code (Rabcd[1], 'myRabcd', 'example-07-rabcd.c', 4)
write_code (Rab[1],   'myRab',   'example-07-rab.c',   2)
```

This creates the following C code for the connection

```
x0 = 1.0/tan(theta);
myGamma [varphi][theta][varphi] = x0;
myGamma [theta][varphi][varphi] = x0;
myGamma [varphi][varphi][theta] = -1.0/2.0*sin(2*theta);
```

Clearly this C-code would not compile (as it stands) for it lacks some basic declarations (e.g., array declarations for myGamma and access to the math library). One solution could be to modify the function write_code to fill in the missing pieces but a better solution is treat the above code as a fragment to be included (either by hand or by a #include) into a separate C-program.

Exercises

7.1. Using the result of Exercise 3.9 (i.e., R_{ab} in terms of g_{ab}) write a Cadabra code that creates C-code that could be the used to compute each of the components of R_{ab}. Assume a generic 3d-metric and assume the coordinates are labelled x,y,z.

Hint: You may need to refer back to Exercise 6.10 to hide the first and second partial derivatives of g_{ab}. You will also need to add simplify=False to the argument list in the call to evaluate.

8 Expressions of interest

A common paradigm in computational science is to break a given problem into smaller parts with each part allocated to a single computer code. This obviously requires some cooperation between the various programs, usually in the form of sharing results – the programs exchange information by exporting and importing data in some suitable format.

Can such a paradigm be applied in the context of Cadabra? Put another way, Is it possible to share Cadabra content between different Cadabra programs? Though this might sound like a simple question it does raise some serious issues. Recall that a Cadabra expression is more than just an object with a list of indices – it may also be subject to a set of properties such as index sets, symmetries, commutation rules etc. Thus when an object is exported the question arises – how much information about its properties should also be exported? And when that object is imported into another program might the inherited properties clash with those declared in the host code? (e.g., an object declared as symmetric in `foo.cdb` might be imported by `bah.cdb` where it is (incorrectly) declared as anti-symmetric).

8.1 Importing Cadabra code

Cadabra allows complete *notebooks* to be imported as a way to share content between sibling codes. For example, suppose `foo.cnb` is a Cadabra notebook with the following content

```
{a,b,c,d,e,f,g,h}::Indices(position=independent).
R_{a b c d}::RiemannTensor.
```

Then another Cadabra program (not necessarily a notebook) can import the above code as in this example

```
from foo import *
expr := R_{a b c d} + R_{a b d c}.
```

The result of this simple example is zero (since $R_{abcd} = -R_a bdc$).

Note the restriction to notebooks - these are written in Cadabra's native format used by the `cadabra2-gtk` gui (though see section 8.3 below for a trick that allows imports from plain files, such as `bah.cdb`).

A good use of this method would be to include all of the common properties from a collection of codes in one notebook. Each code in the collection would then import this one notebook. This saves the programmer time and also ensures consistent definitions across the collection. However, this still leaves the problem of sharing results between one or more programs. One solution, as described in the following section, is to use basic Python IO to read and write the data as required.

8.2 Basic data I/O

The basic idea is to store Cadabra objects as strings in a Python dictionary which in turn is stored as a file in `JSON` format. The code consists of three simple Python functions with the following headers

```
def create (file_name):                    # create a library or clear an exiting library
def put    (key_name,object,file_name): # add an object to the library
def get    (key_name,file_name):           # retrieve an object from the library
```

The implementation of these functions is not all that important here (see the source code in `cadabra2/python` for full details). Note that there are no explicit functions to open or close the library as such actions are handled internally in the `put` and `get` functions.

Here is a simple example that demonstrates the use of these functions. It creates two expressions, writes them to a library, reads them back in but with new names and finally checks that the new objects agree with the originals.

```
lib_name = 'example-08.json'

create (lib_name)

\nabla{#}::Derivative.

gab := g_{a b} - 1/3 x^{c} x^{d} R_{a c b d}
               - 1/6 x^{c} x^{d} x^{e} \nabla_{c}{R_{a d b e}}.

iab := g^{a b} + 1/3 x^{c} x^{d} g^{a e} g^{b f} R_{c e d f}
               + 1/6 x^{c} x^{d} x^{e} g^{a f} g^{b g} \nabla_{c}{R_{d f e g}}.

put ('g_ab',gab,lib_name)
put ('g^ab',iab,lib_name)

foo = get ('g_ab',lib_name)
bah = get ('g^ab',lib_name)

tmp := @(gab) - @(foo).
tmp := @(iab) - @(bah).
```

The output from the above code is as follows. First, the two original objects, g_{ab} and g^{ab}, exported to the file `example-08.json`

$$g_{ab}(x) = g_{ab} - \frac{1}{3}x^c x^d R_{acbd} - \frac{1}{6}x^c x^d x^e \nabla_c R_{adbe}$$

$$g^{ab}(x) = g^{ab} + \frac{1}{3}x^c x^d g^{ae} g^{bf} R_{cedf} + \frac{1}{6}x^c x^d x^e g^{af} g^{bg} \nabla_c R_{dfeg}$$

Second, the new objects, $\bar{g}_{ab}$ and $\bar{g}_{ab}$, imported by reading the file `example-08.json`

$$\bar{g}_{ab}(x) = g_{ab} - \frac{1}{3}x^c x^d R_{acbd} - \frac{1}{6}x^c x^d x^e \nabla_c R_{adbe}$$

$$\bar{g}^{ab}(x) = g^{ab} + \frac{1}{3}x^c x^d g^{ae} g^{bf} R_{cedf} + \frac{1}{6}x^c x^d x^e g^{af} g^{bg} \nabla_c R_{dfeg}$$

Finally, here is the proof that the new and old objects agree

$$g_{ab}(x) - \bar{g}_{ab}(x) = 0$$

$$g^{ab}(x) - \bar{g}^{ab}(x) = 0$$

8.3 Importing code from plain text files

Given that import and friends are native Python constructs it is reasonable to expect that Cadabra should be able to import code from plain text files. How can this be achieved? The trick lies in the format of the plain text – it should be written in pure Python. This leads to the obvious question – How can statements unique to Cadabra be rewritten as Python statements? That is the job of the command line tool cadabra2python. It will create a pure Python code from the Cadabra source. Suppose, for example, that the file foo.cdb contains the following statements

```
{a,b,c,d,e,f,g,h}::Indices(position=independent).
R_{a b c d}::RiemannTensor.
```

This is exactly as per the notebook in Example 8.1 above.

The corresponding Python code, foo.py, can be generated using

```
cadabra2python foo.cdb foo.py
```

The output file, foo.py, with some minor handmade changes (as explained below) consists of the following statements

```
1  import cadabra2
2  from cadabra2 import *
3  __cdbkernel__ = cadabra2.__cdbkernel__
4
5  tmp = Indices(Ex(r'{a,b,c,d,e,f,g,h}'), Ex(r'position=independent)') )
6  tmp = RiemannTensor(Ex(r'R_{a b c d}'), Ex(r'') )
```

This file could then be used instead of the notebook as the target of the import statement (as in the second part of Example 8.1 above).

Note that the first three lines in foo.py where added by hand. They are required so that Python knows how to resolve the function calls (to Ex, Indices and RiemannTensor). Lines 1 and 3 are mandatory while line 2 is optional (without it the call to Ex would need to be written as cadabra2.Ex as per standard Python syntax). Lines 5 and 6 have been cleaned up slightly by replacing triple-quotes with single quotes and substituting tmp for __cdbtemp__.

Part 2 Applications

The examples in the first part of this tutorial were chosen to be sufficiently simple so as to allow the reader to easily gain a good understanding of Cadabra. The risk in using such simple examples is that they might convey the (incorrect) notion that Cadabra is suitable only for such simple calculations. The examples in this second part were chosen to dispel such notions – they are non-trivial calculations and demonstrate that Cadabra is more than capable of handling serious computations in general relativity.

9 The Gauss equation

In this example Cadabra will be used to derive the Gauss equation which relates the induced and ambient curvatures of a hypersurface in an $n-$dimensional Riemannian manifold.

The basics of the underlying mathematics are as follows. Suppose Σ is an $(n-1)-$dimensional subspace of an $n-$dimensional space M. Suppose M is equipped with Riemannian metric g and a metric compatible derivative operator ∇. The subspace Σ will, by way of its embedding in M, inherit a metric and derivative operator which will be denoted by h and D respectively (note this use of D differs from that in the previous sections, here D is a differential operator). Let n^a be the oriented unit normal to Σ. Then the metrics of Σ and M are related by

$$g_{ab} = h_{ab} + n_a n_b$$

while, for any dual-vector v_a lying in Σ (i.e., $v_a n^a = 0$),

$$D_b v_a = h^d{}_b h^c{}_a \nabla_d v_c$$

where $h^a{}_b = g^a{}_b - n^a n_b$ is the projection operator. The curvature tensor for (Σ, h, D) can then be obtained by computing $(D_c D_b - D_b D_c)\, v_a$. This is all very standard and can be found in most textbooks on differential geometry (see [12]).

Translating these equations into Cadabra code is very straightforward and follows a now familiar pattern. Unlike the previous examples, the discussion will begin by considering the fragments of code needed to express the basic mathematical relations as just given. These code fragments will later be glued together to form a complete Cadabra program.

Consider the definition of the projection operator $h^a{}_b = g^a{}_b - n^a n_b$ and its use in defining D in terms of ∇. The symbol `hab` will be used to record the projection operator and `vpq` to record the covariant derivative $D_q v_p$. Thus the code will contain the lines

```
hab:=h^{a}_{b} -> g^{a}_{b} - n^{a} n_{b}:
vpq:=v_{p q} -> h^{a}_{p} h^{b}_{q} \nabla_{b}{v_{a}}:
```

The code will also need an expression for the commutation of the covariant derivatives, $(D_r D_q - D_q D_r)\, v_p$ which will be written as `vpqr`

```
vpqr:=h^{a}_{p} h^{b}_{q} h^{c}_{r} ( \nabla_{c}{v_{a b}} - \nabla_{b}{v_{a c}} ).

substitute (vpq,hab)
substitute (vpqr,vpq)
```

The code will also need to contain some standard substitutions to simplify and tidy the result. Note that all of the previous definitions and the following substitutions are exactly what would normally be done if these calculations were done by hand. For example, the lines

```
substitute (vpqr,$h^{a}_{b} n^{b} -> 0$)
substitute (vpqr,$h^{a}_{b} n_{a} -> 0$)
```

expresses the condition that n^a is normal to the subspace, $0 = n^b h^a{}_b$ and $0 = n^b h_b{}^a$. The line

```
substitute (vpqr,$\nabla_{a}{g^{b}_{c}} -> 0$)
```

states that the covariant derivative of g is zero while the line

```
substitute (vpqr,$n^{a} \nabla_{b}{v_{a}} -> -v_{a} \nabla_{b}{n^{a}}$)
```

is a simple re-working of $0 = \nabla\left(n^a v_a\right) = \left(\nabla n^a\right) v_a + n^a \left(\nabla v_a\right)$ to eliminate first derivatives of v^a from the expression `vpqr`. The next line

```
substitute (vpqr,$v_{a} \nabla_{b}{n^{a}} -> v_{p} h^{p}_{a} \nabla_{b}{n^{a}}$)
```

squeezes a projection operator between v_a and ∇n^a. This is allowed because v^a has zero normal component. Finally, lines like

```
substitute (vpqr,$h^{p}_{a} h^{q}_{b} \nabla_{p}{n_{q}} -> K_{a b}$)
substitute (vpqr,$h^{p}_{a} h^{q}_{b} \nabla_{p}{n^{b}} -> K_{a}^{q}$)
```

can be used to introduce the extrinsic curvature tensor K_{ab}.

The above code fragments will need to be supplemented with extra statements, such as an index set, substitution and simplification rules etc., before Cadabra can do its job. Such pieces of code are very similar to those given in the previous examples and thus require no further explanation here. Here then is the final code.

```
{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u#}::Indices(position=independent).

\nabla{#}::Derivative.

K_{a b}::Symmetric.
g^{a}_{b}::KroneckerDelta.

# define the projection operator

hab:=h^{a}_{b} -> g^{a}_{b} - n^{a} n_{b}.

# 3-covariant derivative obtained by projection on 4-covariant derivative

vpq:=v_{p q} -> h^{a}_{p} h^{b}_{q} \nabla_{b}{v_{a}}.

# compute 3-curvature by commutation of covariant derivatives

vpqr:= h^{a}_{p} h^{b}_{q} h^{c}_{r} (  \nabla_{c}{v_{a b}}
                                       - \nabla_{b}{v_{a c}} ).

substitute (vpq,hab)
substitute (vpqr,vpq)

distribute   (vpqr)
product_rule (vpqr)
distribute   (vpqr)
eliminate_kronecker (vpqr)

# standard substitutions

substitute (vpqr,$h^{a}_{b} n^{b} -> 0$)
substitute (vpqr,$h^{a}_{b} n_{a} -> 0$)
substitute (vpqr,$\nabla_{a}{g^{b}_{c}} -> 0$)
substitute (vpqr,$n^{a} \nabla_{b}{v_{a}} -> -v_{a} \nabla_{b}{n^{a}}$)
substitute (vpqr,$v_{a} \nabla_{b}{n^{a}} -> v_{p} h^{p}_{a}\nabla_{b}{n^{a}}$)
substitute (vpqr,$h^{p}_{a} h^{q}_{b} \nabla_{p}{n_{q}} -> K_{a b}$)
substitute (vpqr,$h^{p}_{a} h^{q}_{b} \nabla_{p}{n^{b}} -> K_{a}^{q}$)

# tidy up

{h^{a}_{b},\nabla_{a}{v_{b}}}::SortOrder.

sort_product   (vpqr)
rename_dummies (vpqr)
canonicalise   (vpqr)
factor_out     (vpqr,$h^{a?}_{b?}$)
factor_out     (vpqr,$v_{a?}$)
```

At this stage Cadabra's output is

$$(D_r D_q - D_q D_r)v_p = h^a{}_p h^b{}_q h^c{}_r \left(\nabla_c \left(\nabla_b v_a\right) - \nabla_b \left(\nabla_c v_a\right)\right) + v_a \left(K_q{}^a K_{pr} - K_r{}^a K_{pq}\right) \qquad /47/$$

which, although correct, is not in the familiar textbook form. This minor quibble is easily addressed by making good use of the results from the previous example. Thus, the respective Riemann tensors for the metrics g and h can be written as

$$(D_r D_q - D_q D_r)v_p = {}^h R^a{}_{pqr} v_a$$
$$(\nabla_r \nabla_q - \nabla_q \nabla_r)v_p = {}^g R^a{}_{pqr} v_a$$

These relations, along with the simple observations that $v_a = h^b{}_a v_b$ and $K_a{}^b = h^b{}_c K_a{}^c$, can be used to massage Cadabra's output into the familiar textbook form. The Cadabra code for this final stage is

```
R{#}::LaTeXForm("{{\strut}^g R}").

gRabcd := \nabla_{c}{\nabla_{b}{v_{a}}}
         -\nabla_{b}{\nabla_{c}{v_{a}}} -> R^{d}_{a b c} v_{d}.

substitute     (vpqr,gRabcd)
distribute     (vpqr)
substitute     (vpqr,$v_{a} -> h^{b}_{a} v_{b}$)
substitute     (vpqr,$h^{b}_{a} K_{c}^{a} -> K_{c}^{b}$)
sort_product   (vpqr)
rename_dummies (vpqr)
canonicalise   (vpqr)
factor_out     (vpqr,$v_{a?}$)
substitute     (vpqr,$v_{a}->1$)
sort_product   (vpqr)
```

The final output is now

$${}^h R^a{}_{pqr} = h^a{}_e h^b{}_p h^c{}_q h^d{}_r \, {}^g R^e{}_{bcd} + K_q{}^a K_{pr} - K_r{}^a K_{pq} \qquad /62/$$

which is the standard textbook form for the Gauss equation.

10 The metric determinant in Riemann normal coordinates

This and the following two examples are based on the standard leading order expansion of a metric in Riemann normal coordinates, namely

$$\begin{aligned} g_{ab}(x) = g_{ab} - \frac{1}{3}x^c x^d R_{acbd} - \frac{1}{6}x^c x^d x^e \nabla_c R_{adbe} \\ + \frac{1}{180}x^c x^d x^e x^f \left(8g^{gh}R_{acdg}R_{befh} - 9\nabla_{cd}R_{aebf}\right) + \cdots \end{aligned} \tag{10.1}$$

and

$$\begin{aligned} g^{ab}(x) = g^{ab} + \frac{1}{3}x^c x^d g^{ae} g^{bf} R_{cedf} + \frac{1}{6}x^c x^d x^e g^{af} g^{bg} \nabla_c R_{dfeg} \\ + \frac{1}{60}x^c x^d x^e x^f g^{ag} g^{bh} \left(4g^{ij}R_{cgdi}R_{ehfj} + 3\nabla_{cd}R_{egfh}\right) + \cdots \end{aligned} \tag{10.2}$$

where g_{ab}, g^{ab}, R_{abcd} and its derivatives are evaluated at the origin. The expansions are valid inside a suitably chosen neighbourhood of the origin. Note that it is customary to choose $g_{ab} = \text{diag}(-1,1,1,1)$ for Lorentzian spacetimes. For more details on Riemann normal coordinates, their derivation and use, see [13], [12], [14], [15], [16]. See also the website https://github.com/leo-brewin/riemann-normal-coords/ for an extensive set of Cadabra programs for developing Riemann normal expansions (this includes the code used to generate the above expressions for the metric and its inverse).

The above expansions includes terms up-to fourth order in the coordinates. Thus when forming other quantities based on this metric, such as the metric determinant (in this example) and the connection (the next two examples), care must be taken to truncate the computed expression to fourth order (at most).

The metric determinant can be easily computed using a basic result from linear algebra. For the 4×4 matrices N and M built from N^{ab} and M_{ab} then

$$\det N \det M = \frac{1}{4!}\epsilon^{abcd}_{pqrs} M_{ia}M_{jb}M_{kc}M_{ld}N^{ip}N^{jq}N^{kr}N^{ls}$$

where $\det N = \det(N^{ab})$ and $\det M = \det(M_{ij})$.

Choosing $M_{ab} = g_{ab}(x)$ and $N^{ab} = g^{ab} = \text{diag}(-1,1,1,1)$ leads to the following simple expression for the metric determinant

$$\det g(x) = -\frac{1}{4!}\,\epsilon^{abcd}_{pqrs}\, g_{ia}(x)\, g_{jb}(x)\, g_{kc}(x)\, g_{ld}(x) g^{ip} g^{jq} g^{kr} g^{ls}$$

Implementing the above in Cadabra is quite straightforward though there a two minor points worth noting. First, Cadabra provides an algorithm, `asym`, that can be used to impose anti-symmetry on chosen objects. This can be used to create ϵ^{abcd}_{pqrs} using code similar to

```
d{#}::KroneckerDelta.
eps := d^{a}_{p} d^{b}_{q} d^{c}_{r} d^{d}_{s}.
asym (eps, $^{a}, ^{b}, ^{c}, ^{d}$)
```

The first line declares d to be a Kronecker delta while the second creates a seed for ϵ. The call to asym uses this seed to create a new object that is antisymmetric in the nominated indices (i.e., the upper indices $(abcd)$). The bonus in using asym is that it will include the $1/4!$ factor, that is, asym returns $(1/4!\,)\epsilon^{abcd}_{pqrs}$.

The second point concerns the truncation of $\det g$ to be consistent with that of the metric (in this case to fourth order). The question is – at what stage in the computation should the truncation be imposed? The answer will have a significant impact on the computational cost (particularly for higher order expansions). The lazy approach is to defer the truncation until the very end. For a fourth order expansion in four dimensions this would produce a polynomial of order $4^4 = 256$ and yet only the first four terms would be retained. This is a huge waste of resources. A better approach would be to compute the terms in $\det g$ in successive orders, starting from zeroth order. One way to do so would be to first decompose $g_{ab}(x)$ into successive orders,

$$g_{ab}(x) = \overset{0}{g}_{ab} + \overset{2}{g}_{ab} + \overset{3}{g}_{ab} + \overset{4}{g}_{ab} + \cdots$$

where $\overset{n}{g}$ denotes the n-th order term of $g_{ab}(x)$. A similar expansion can be proposed for $\det g$, namely,

$$\det g = \overset{0}{\det}\, g + \overset{2}{\det}\, g + \overset{3}{\det}\, g + \overset{4}{\det}\, g + \cdots$$

A standard procedure of substitution, expansion and matching can then be applied. The result would be a series of equations that would allow, for example, $\overset{0}{\det}\, g$ to be computed from $\overset{0}{g}_{ab}$, $\overset{1}{\det}\, g$ to be computed form $\overset{0}{g}_{ab}$ and $\overset{1}{g}_{ab}$ etc.

Despite the clear advantage of this second scheme the code given in the cadabar2/examples/ directory uses the previous lazy method. Why? For the simple reason that it was easy to write and it gave results in a reasonable time (similar to most of the other codes in this tutorial). It is certain that this lazy code will be too expensive for higher order expansions (or in higher dimensions).

The actual computation of $\det g$ requires only a few lines of Cadabra code

```
# compute Ndetg = negative det g
Ndetg := @(eps) gx_{p a} gx_{q b} gx_{r c} g^{i p} g^{j q} g^{k r}.
substitute       (Ndetg,gxab)
distribute       (Ndetg)
Ndetg = truncate (Ndetg,4)
substitute       (Ndetg,$g^{a b} g_{b c} -> d^{a}_{c}$,repeat=True)
eliminate_kronecker (Ndetg)
```

where gx_{a b} represents the metric $g_{ab}(x)$ given above and gxab is a rule that substitutes $g_{ab}(x)$ for gx_{a b}.

The remainder of the code is just housekeeping in particular the introduction of the Ricci tensor

```
substitute (Ndetg,$R_{a b c d} g^{a c}                 -> R_{b d}$,repeat=True)
substitute (Ndetg,$\nabla_{a}{R_{b c d e}} g^{b d}    -> \nabla_{a}{R_{c e}}$,repeat=True)
substitute (Ndetg,$\nabla_{a b}{R_{c d e f}} g^{c e} -> \nabla_{a b}{R_{d f}}$,repeat=True)
```

The final result for $\det g$, to fourth order in x^a, is

$$\begin{aligned}-\det g(x) = 1 &- \frac{1}{3}x^a x^b R_{ab} - \frac{1}{6}x^a x^b x^c \nabla_a R_{bc} \\ &+ \frac{1}{180}x^a x^b x^c x^d \left(-9\nabla_{ab}R_{cd} + 10R_{ab}R_{cd} - 2g^{ef}g^{gh}R_{aebg}R_{cfdh}\right) + \cdots\end{aligned}$$

11 The metric connection in Riemann normal coordinates

Though the subject of this example will be the computation of the connection for the metric in Riemann normal form most of the discussion will concern the computational costs. These costs will increase with higher order expansions. The surprising thing is just how easy it is to hit the computational wall. Fortunately (for this example) there are simple ways to manage this problem.

The starting point is the familiar equation for the connection

$$\Gamma^{a}{}_{bc}(x) = \frac{1}{2} g^{ad} \left(\partial_b g_{dc} + \partial_c g_{bd} - \partial_d g_{bc} \right)$$

where $g_{ab}(x)$ and $g^{ab}(x)$ are given by (10.1) and (10.2) respectively. The proliferation of terms will arise first through the product rule acting on the individual terms and second through the expansion of $g^{ab}(x)$ and its coupling with the derivative terms. It is also clear that at some point the result will need to be truncated to an order consistent with that of the metric.

An obvious strategy (to minimise computational cost) is to avoid introducing unnecessary terms. One clear case of this occurs when computing the derivatives of terms such as $\partial_a R_{bcde} x^x x^e$. Since the R_{abcd} are constants (evaluated at the origin of the RNC) it follows that

$$\partial_a R_{bcde} x^x x^e = R_{bcde} \partial_a x^x x^e \tag{11.1}$$

How is this simple step implemented in Cadabra? One (naive) option is invoke a product rule then set the derivatives of R_{abcd} to zero. A better option is to inform Cadabra ahead of time which objects are non-constant. Here is a short fragment that does the job.

```
1  \partial{#}::PartialDerivative.
2  x{#}::Depends{\partial}.
3
4  term := \partial_{a}{R_{bcde} x^{x} x^{e}}.
5
6  unwrap (term)
```

The idea is to identify the objects that will survive under the action of a derivative operator. This can be seen in line 2 where x^a is explicitly declared to depend on `\partial`. This information is used by `unwrap` to pull out factors that do *not* depend on the derivative operators. Thus in line 6 the R_{abcd} will be pulled out as a common factor since they were not declared to depend upon `\partial`. Consequently, the value of `term`, after the call to `unwrap`, will equal that of the right hand side of equation (11.1). At this point the computation can proceed by invoking a product rule to reduce $\partial_a(x^x x^e)$ to Kronecker deltas.

The other bits of Cadabra required to complete the computation include rules to define the metric, the connection and, after the main body of the code, some basic housekeeping. The main body of the code (excluding the rules for the metric and its inverse) is

```
ChrSym := \Gamma^{a}_{b c} -> 1/2 g^{a d} (  D_{b}{g_{d c}}
                                           + D_{c}{g_{b d}}
                                           - D_{d}{g_{b c}} ).


Gamma := \Gamma^{a}_{b c}.
```

```
substitute        (Gamma,ChrSym)   # the connection
substitute        (Gamma,gab)      # the metric
substitute        (Gamma,iab)      # the metric inverse
distribute        (Gamma)
unwrap            (Gamma)          # pull out constants
product_rule      (Gamma)
distribute        (Gamma)
substitute        (Gamma,Dx)       # rule for partial derivs of x
eliminate_kronecker (Gamma)
```

At this point the expression for $\Gamma^a{}_{bc}(x)$ will contain terms beyond 3rd-order[c] in x^a. A key question is then – when should this truncation be imposed? A good choice is to truncate *before* applying the housekeeping as in the following code

```
Gamma = truncate (Gamma,3)

sort_product    (Gamma)
rename_dummies  (Gamma)
canonicalise    (Gamma)
```

The code for `truncate` is similar to that used in Example 4. This works well and produces the following result

$$\begin{aligned}\Gamma^a{}_{bc}(x) = {} & \frac{1}{3} g^{ad} x^e \left(R_{bdce} + R_{becd}\right) \\ & + \frac{1}{12} g^{ad} x^e x^f \left(-\nabla_c R_{bedf} + \nabla_d R_{becf} + 2\nabla_e R_{bdcf} + 2\nabla_e R_{bfcd} - \nabla_b R_{cedf}\right) \\ & + \frac{1}{40} g^{ad} x^e x^f x^g \left(-\nabla_{ce} R_{bfdg} - \nabla_{ec} R_{bfdg} + \nabla_{de} R_{bfcg} + \nabla_{ed} R_{bfcg} + 2\nabla_{ef} R_{bdcg}\right. \\ & \quad \left. + 2\nabla_{ef} R_{bgcd} - \nabla_{be} R_{cfdg} - \nabla_{eb} R_{cfdg}\right) + \frac{1}{45} g^{ad} g^{ef} x^g x^h x^i \left(4R_{becg} R_{dhfi} + 4R_{bgce} R_{dhfi}\right. \\ & \quad \left. - 2R_{bdeg} R_{chfi} - R_{bedg} R_{chfi} + R_{bgde} R_{chfi} - 2R_{bgeh} R_{cdfi} - R_{bgeh} R_{cfdi} + R_{bgeh} R_{cidf}\right)\end{aligned}$$

It was previously noted that it is rather easy to hit the computational wall. Here is a slightly changes that does just that – truncate the result *after* the housekeeping, that is

```
sort_product    (Gamma)
rename_dummies  (Gamma)
canonicalise    (Gamma)

Gamma = truncate (Gamma,3)
```

This code was terminated (by hand) with no results after running for over 20 minutes and using over 500 Mbytes of memory. In contrast the previous code completed in around 33 seconds and required 60 Mbytes of memory.

By conducting a few experiments it was found that the slow code stalled on the call to `canonicalise`. The problem here is that `canonicalise` is being asked to do its magic across all of the terms in `Gamma` which for a 4th-order metric is approximately 200 terms. And since `Gamma` is a polynomial in x^a the housekeeping (sort, rename, canonicalise) will naturally target each

[c]Why focus on 3rd order? Because the metric and its inverse are known only to 4th-order and the $\Gamma^a{}_{bc}(x)$ requires one derivative in x^a.

power of x^a. Thus a better approach would be to decompose `Gamma` into separate powers of x^a, do the housekeeping on each power then rebuild `Gamma`. That will work but it is a silly solution as there is no point in doing any housekeeping on the higher order terms as they are discarded in the later call to `truncate`. The main point in this variation is to show how simple changes (without thought) can dramatically blow out the computational cost. The advice given earlier, to keep as few terms as needed, is well worth remembering.

12 The third order terms of Calzetta etal.

The metric connection $\Gamma^a{}_{bc}(x)$ is symmetric in its lower indices. Thus there is no loss of information in forming a product like $z^b z^c \Gamma^a{}_{bc}(x)$. The z^a have no real meaning, they are just to help with the bookkeeping. Now define Γ^a by

$$\Gamma^a := z^b z^c \Gamma^a{}_{bc}(x)$$

then using the results from the previous example it is easy to show that

$$\begin{aligned}
\Gamma^a &= z^b z^c \Gamma^a{}_{bc}(x)\\
&= \frac{2}{3} x^b z^c z^d R_{acbd} + \frac{1}{12} x^b x^c z^d z^e \left(4\nabla_b R_{adce} + 2\nabla_d R_{abce} + \nabla_a R_{bdce}\right)\\
&\quad + \frac{1}{40} x^b x^c x^d z^e z^f \left(4\nabla_{bc} R_{aedf} + 2\nabla_{be} R_{acdf} + 2\nabla_{eb} R_{acdf} + \nabla_{ab} R_{cedf} + \nabla_{ba} R_{cedf}\right)\\
&\quad + \frac{2}{45} g^{bc} x^d x^e x^f z^g z^h \left(4R_{adbe} R_{cgfh} - 2R_{agbd} R_{cefh} - R_{adbg} R_{cefh} + R_{abdg} R_{cefh}\right)
\end{aligned}$$

Calzetta etal.[17] have also computed an expression for the connection in Riemann normal coordinates. Their result, denoted by $\bar\Gamma$,

$$\begin{aligned}
\bar\Gamma^\mu &= z^\nu z^\rho \bar\Gamma^\mu{}_{\nu\rho}(x)\\
&= z^\nu z^\rho \Big\{ \frac{2}{3} R^\mu{}_{\nu\rho\sigma} x^\sigma + \frac{1}{12}\left(5\nabla_\lambda R^\mu{}_{\nu\rho\sigma} + \nabla_\rho R^\mu{}_{\sigma\nu\lambda}\right) x^\sigma x^\lambda\\
&\qquad + \frac{1}{6}\Big[\frac{9}{10}\nabla_{\tau\lambda} R^\mu{}_{\rho\nu\sigma} + \frac{3}{20}\left(\nabla_{\tau\rho} R^\mu{}_{\sigma\nu\lambda} + \nabla_{\rho\tau} R^\mu{}_{\sigma\nu\lambda}\right)\\
&\qquad\qquad + \frac{1}{60}\left(21 R^\mu{}_{\lambda\xi\rho} R^\xi{}_{\sigma\nu\tau} + 48 R^\mu{}_{\xi\rho\lambda} R^\xi{}_{\sigma\nu\tau} - 37 R^\mu{}_{\sigma\xi\lambda} R^\xi{}_{\nu\rho\tau}\right)\Big] x^\sigma x^\lambda x^\tau \Big\}
\end{aligned}$$

appears (at first sight) to differ significantly from Γ. The purpose of this example is to show that both expressions agree. The basic approach will be to compute the difference $\Delta\Gamma := \Gamma - \bar\Gamma$ and to then use the known symmetries of the Riemann tensor to show that all terms cancel.

Note that the convention for the Riemann tensor used by Calzetta etal. is opposite to that used in this tutorial. This is easily accounted for (in the Cadabra code) by replacing their R_{abcd} with $-R_{abcd}$.

Other simple changes will also be made to Γ and $\bar\Gamma$ before attempting to show that $\Gamma - \bar\Gamma = 0$. One obvious change is that a single index set should be used for both Γ and $\bar\Gamma$. Further changes include lowering indices so that each Riemann term is of the form R_{abcd} and sorting all products into a consistent order $g, x, z, R, \nabla R, \nabla\nabla R$. The goal in making these changes is simply to maximise the opportunity to use known Riemann symmetries and to spot the terms that cancel. These basic preconditioning steps are loosely implemented in Cadabra as follows.

Converting the Greek indices on $\bar\Gamma$ to Latin indices can be done in Cadabra by first declaring a named pair of index sets

```
{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,u,v#}::Indices("latin",position=independent).
{\mu,\nu,\rho,\sigma,\tau,\lambda,\xi#}::Indices("greek",position=independent).
```

and then passing this pair to `rename_dummies` as in

```
rename_dummies (\GammaBar,"greek","latin")
```

The call to rename_dummies only renames the dummy indices (now that was a surprise). This leaves the free index μ on $\bar{\Gamma}^\mu$ unchanged. This little problem can be dealt with by lowering the index using $\delta_{a\mu}\bar{\Gamma}^\mu$ and then eliminating the Kronecker deltas

```
\delta{#}::KroneckerDelta.
GammaBar := \delta_{a \mu} @(GammaBar).
distribute (GammaBar)
eliminate_kronecker (GammaBar)
```

Note that a small liberty has been take here – the index lowering should be done using $g_{a\mu}$ rather than $\delta_{a\mu}$. But the nett outcome is the same and it saves having to include extra code to implement the action of $g_{a\mu}$ on each term (the call to eliminate_kronecker does the same for $\delta_{a\mu}$).

Lowering the upper index on Γ^a is slightly more involved as shown is this Cadabra fragment

```
# lower free index ^{a} to _{v}

Gamma := g_{v a} @(Gamma).

distribute (Gamma)
substitute (Gamma, $g_{a d} g^{d b} -> \delta_{a}^{b}$)
eliminate_kronecker (Gamma)

# change free index _{v} to _{a}

foo := tmp_{v} -> @(Gamma).
bah := tmp_{a}.
substitute (bah, foo)

Gamma := @(bah).
```

This involves two steps. First, lower the index a and covert it to v. Second, convert the index v back to a. Note that this second step could also be implemented in Cadabra using

```
Gamma := \delta^{v}_{a} @(Gamma).
distribute (Gamma)
eliminate_kronecker (Gamma)
```

Recall that Cadabra will do the necessary index juggling to avoid any clash that might arise in the above computation (see the results of Exercise 1.9).

At this point the difference $\Delta\Gamma_a := \Gamma_a - \bar{\Gamma}_a$ is given by

$$\begin{aligned}\Delta\Gamma_a &= \frac{1}{12}x^b x^c z^d z^e \left(\nabla_d R_{abce} - \nabla_b R_{adce} + \nabla_a R_{bdce}\right)\\ &+ \frac{1}{40}x^b x^c x^d z^e z^f \left(\nabla_{be} R_{acdf} + \nabla_{eb} R_{acdf} - 2\nabla_{bc} R_{aedf} + \nabla_{ab} R_{cedf} + \nabla_{ba} R_{cedf}\right)\\ &+ \frac{1}{360}g^{bc}x^d x^e x^f z^g z^h \left(-32R_{abdg}R_{cefh} + 27R_{adbe}R_{cgfh} + 5R_{adbg}R_{cefh} - 32R_{agbd}R_{cefh}\right)\end{aligned}$$

Now the fun begins (cancelling terms). First notice that $\Delta\Gamma$ consists of second and third order terms in x. It is easy to see that the second order terms will vanish when the second Bianchi identity is applied. The third order terms require a little more work. The first step is to commute the order of the second covariant derivatives on the ∇_{eb} and ∇_{ab} terms. This of course will introduce new RR terms which couple with the existing RR terms.

Each of these steps can be implemented in Cadabra by applying suitable substitution rules on a `zoom`'ed and tagged expression (along the lines shown in Example 5). For example, the following code applies the second Bianchi identity to the second order terms

```
 1  diff2 = get_xterm (diff,2)
 2  diff3 = get_xterm (diff,3)
 3
 4  diff2 = add_tags (diff2,'\\mu')
 5
 6  # swap indices on middle term, then apply 2nd Bianchi identity
 7
 8  zoom       (diff2, $\mu_{1} Q??$)
 9  substitute (diff2, $\nabla_{b}{R_{a d c e}} -> - \nabla_{b}{R_{d a c e}}$)
10  unzoom     (diff2)
11
12  substitute (diff2, $\mu_{1} -> \mu_{0}, \mu_{2} -> \mu_{0}$)
13  substitute (diff2, $\mu_{0} -> 0$)
14
15  diff2 = clear_tags (diff2,'\\mu')
16
17  diff := @(diff2) + @(diff3).
```

The code is rather easy to understand. The first pair of lines extracts the second and third order terms. The second order terms are then tagged in line 4. Lines 8 to 10 isolates the target (the middle term) and applies the substitution (swapping indices ad on $\nabla_b R_{adce}$). The three terms are united (in line 12) by setting $\mu_0 = \mu_1 = \mu_2$ and then eliminated (in line 13) by setting $\mu_0 = 0$. Finally, the tags are cleared in line 15. This last step is not really needed since `diff2` is zero. The last line of the code rebuilds `diff` for later processing of the third order terms.

Similar code can be used to commute the second covariant derivatives leading to

$$\begin{aligned}\Delta\Gamma_a = {}&\frac{1}{40}x^b x^c x^d z^e z^f \left(2\nabla_{be}R_{acdf} - 2\nabla_{bc}R_{aedf} + 2\nabla_{ba}R_{cedf}\right)\\ &+ \frac{1}{360}g^{bc}x^d x^e x^f z^g z^h \left(-32R_{abdg}R_{cefh} + 32R_{adbg}R_{cefh} - 32R_{agbd}R_{cefh}\right)\end{aligned}$$

The final steps are now rather obvious – apply the second Bianchi identity to the first term and the first Bianchi identity to the second term. These steps are (once again) implemented using code very similar to that given above. The result is that $\Delta\Gamma_a = 0$, that is $\Gamma = \bar{\Gamma}$.

13 The Weyl tensor vanishes in 3-dimensions

The Weyl tensor in an N-dimensional space is given by

$$C_{abcd} = R_{abcd} + \frac{1}{N-2}(R_{ad}g_{bc} - R_{ac}g_{bd} + g_{ad}R_{bc} - g_{ac}R_{bd}) + \frac{R}{(N-1)(N-2)}(g_{ac}g_{bd} - g_{ad}g_{bc})$$

The $C^a{}_{bcd}$ shares not only all of the symmetries of the Riemann tensor but it also satisfies $C^a{}_{bad} = 0$. Thus the number $M(N)$ of algebraically independent components of $C^a{}_{bcd}$ at a point in an N dimensional space is given by

$$M(N) = \frac{N^2(N^2-1)}{12} - \frac{N(N+1)}{2}$$

A common argument is that since $M(3) = 0$ it follows that the Weyl tensor has zero algebraically independent components and thus must vanish in 3 dimensions. It should also be possible to establish the same result by direct computation, that is, show that $C^a{}_{bad} = 0$ for any Riemannian metric in a 3 dimensional space. That is the aim of this example. Two methods will be presented. The first uses a brute force method where the Weyl tensor is evaluated on a generic 3-metric. The second method uses only the known symmetries of the Riemann tensor to show that all frame components of the Weyl tensor are zero (and thus that the Weyl tensor must also be zero).

13.1 Proof by brute force

The computational steps are quite straightforward. First start with basic definitions for the connection, the Riemann, Ricci and Weyl tensors.

```
GammaU := \Gamma^{a}_{b c} ->  1/2 g^{a d} (   \partial_{b}{g_{d c}}
                                             + \partial_{c}{g_{b d}}
                                             - \partial_{d}{g_{b c}}).

GammaD := \Gamma_{a b c} ->  1/2 (   \partial_{b}{g_{a c}}
                                   + \partial_{c}{g_{b a}}
                                   - \partial_{a}{g_{b c}}).

Rabcd := R_{a b c d} ->   \partial_{c}{\Gamma_{a b d}}
                        - \partial_{d}{\Gamma_{a b c}}
                        + \Gamma_{e a d} \Gamma^{e}_{b c}
                        - \Gamma_{e a c} \Gamma^{e}_{b d}.

Rab := R_{a b} -> g^{c d} R_{a c b d}.

Rscalar := R -> g^{a b} R_{a b}.

# Weyl in 3-dimensions

Cabcd := R_{a b c d} - (R_{a c} g_{b d} - R_{a d} g_{b c})
                     - (g_{a c} R_{b d} - g_{a d} R_{b c})
               + 1/2 R (g_{a c} g_{b d} - g_{a d} g_{b c}).
```

Then combine these into a single expression for the Weyl tensor expressed solely in terms of the metric.

```
substitute      (Cabcd,Rscalar)
substitute      (Cabcd,Rab)
substitute      (Cabcd,Rabcd)
substitute      (Cabcd,GammaU)
substitute      (Cabcd,GammaD)

distribute      (Cabcd)

sort_product    (Cabcd)
rename_dummies  (Cabcd)
canonicalise    (Cabcd)
```

The final step is to evaluate this expression on a generic metric in 3-dimensions.

```
gab := {g_{x x} = gxx, g_{x y} = gxy, g_{x z} = gxz,
        g_{y x} = gxy, g_{y y} = gyy, g_{y z} = gyz,
        g_{z x} = gxz, g_{z y} = gyz, g_{z z} = gzz}.

complete (gab, $g^{a b}$)
evaluate (Cabcd,gab)
```

The result is that the Weyl tensor is zero (as expected).

$$
\begin{aligned}
8C_{abcd} &= 8R_{abcd} - 8R_{ac}g_{bd} + 8R_{ad}g_{bc} - 8g_{ac}R_{bd} + 8g_{ad}R_{bc} + 4R\left(g_{ac}g_{bd} - g_{ad}g_{bc}\right) && /22/\\
&= 4\partial_{bc}g_{ad} - 4\partial_{ac}g_{bd} - 4\partial_{bd}g_{ac} + 4\partial_{ad}g_{bc} + 2\partial_a g_{de}\partial_b g_{cf}g^{ef} + 2\partial_a g_{de}\partial_c g_{bf}g^{ef}\\
&\quad - 2\partial_a g_{de}\partial_f g_{bc}g^{ef} + 2\partial_b g_{ce}\partial_d g_{af}g^{ef} + 2\partial_c g_{be}\partial_d g_{af}g^{ef} - 2\partial_d g_{ae}\partial_f g_{bc}g^{ef} - 2\partial_b g_{ce}\partial_f g_{ad}g^{ef}\\
&\quad - 2\partial_c g_{be}\partial_f g_{ad}g^{ef} + 2\partial_e g_{ad}\partial_f g_{bc}g^{ef} - 2\partial_a g_{ce}\partial_b g_{df}g^{ef} - 2\partial_a g_{ce}\partial_d g_{bf}g^{ef} + 2\partial_a g_{ce}\partial_f g_{bd}g^{ef}\\
&\quad - 2\partial_b g_{de}\partial_c g_{af}g^{ef} - 2\partial_c g_{ae}\partial_d g_{bf}g^{ef} + 2\partial_c g_{ae}\partial_f g_{bd}g^{ef} + 2\partial_b g_{de}\partial_f g_{ac}g^{ef} + 2\partial_d g_{be}\partial_f g_{ac}g^{ef}\\
&\quad - 2\partial_e g_{ac}\partial_f g_{bd}g^{ef} - 4\partial_{ce}g_{af}g_{bd}g^{ef} + 4\partial_{ac}g_{ef}g_{bd}g^{ef} + 4\partial_{ef}g_{ac}g_{bd}g^{ef} - 4\partial_{ae}g_{cf}g_{bd}g^{ef}\\
&\quad - 2\partial_a g_{ef}\partial_c g_{gh}g_{bd}g^{eg}g^{fh} - 4\partial_e g_{af}\partial_g g_{ch}g_{bd}g^{eg}g^{fh} + 4\partial_e g_{af}\partial_g g_{ch}g_{bd}g^{eh}g^{fg}\\
&\quad + 4\partial_a g_{ce}\partial_f g_{gh}g_{bd}g^{eg}g^{fh} - 2\partial_a g_{ce}\partial_f g_{gh}g_{bd}g^{ef}g^{gh} + 4\partial_c g_{ae}\partial_f g_{gh}g_{bd}g^{eg}g^{fh}\\
&\quad - 2\partial_c g_{ae}\partial_f g_{gh}g_{bd}g^{ef}g^{gh} - 4\partial_e g_{ac}\partial_f g_{gh}g_{bd}g^{eg}g^{fh} + 2\partial_e g_{ac}\partial_f g_{gh}g_{bd}g^{ef}g^{gh} + 4\partial_{de}g_{af}g_{bc}g^{ef}\\
&\quad - 4\partial_{ad}g_{ef}g_{bc}g^{ef} - 4\partial_{ef}g_{ad}g_{bc}g^{ef} + 4\partial_{ae}g_{df}g_{bc}g^{ef} + 2\partial_a g_{ef}\partial_d g_{gh}g_{bc}g^{eg}g^{fh}\\
&\quad + 4\partial_e g_{af}\partial_g g_{dh}g_{bc}g^{eg}g^{fh} - 4\partial_e g_{af}\partial_g g_{dh}g_{bc}g^{eh}g^{fg} - 4\partial_a g_{de}\partial_f g_{gh}g_{bc}g^{eg}g^{fh}\\
&\quad + 2\partial_a g_{de}\partial_f g_{gh}g_{bc}g^{ef}g^{gh} - 4\partial_d g_{ae}\partial_f g_{gh}g_{bc}g^{eg}g^{fh} + 2\partial_d g_{ae}\partial_f g_{gh}g_{bc}g^{ef}g^{gh}\\
&\quad + 4\partial_e g_{ad}\partial_f g_{gh}g_{bc}g^{eg}g^{fh} - 2\partial_e g_{ad}\partial_f g_{gh}g_{bc}g^{ef}g^{gh} - 4\partial_{de}g_{bf}g_{ac}g^{ef} + 4\partial_{bd}g_{ef}g_{ac}g^{ef}\\
&\quad + 4\partial_{ef}g_{bd}g_{ac}g^{ef} - 4\partial_{be}g_{df}g_{ac}g^{ef} - 2\partial_b g_{ef}\partial_d g_{gh}g_{ac}g^{eg}g^{fh} - 4\partial_e g_{bf}\partial_g g_{dh}g_{ac}g^{eg}g^{fh}\\
&\quad + 4\partial_e g_{bf}\partial_g g_{dh}g_{ac}g^{eh}g^{fg} + 4\partial_b g_{de}\partial_f g_{gh}g_{ac}g^{eg}g^{fh} - 2\partial_b g_{de}\partial_f g_{gh}g_{ac}g^{ef}g^{gh}\\
&\quad + 4\partial_d g_{be}\partial_f g_{gh}g_{ac}g^{eg}g^{fh} - 2\partial_d g_{be}\partial_f g_{gh}g_{ac}g^{ef}g^{gh} - 4\partial_e g_{bd}\partial_f g_{gh}g_{ac}g^{eg}g^{fh}\\
&\quad + 2\partial_e g_{bd}\partial_f g_{gh}g_{ac}g^{ef}g^{gh} + 4\partial_{ce}g_{bf}g_{ad}g^{ef} - 4\partial_{bc}g_{ef}g_{ad}g^{ef} - 4\partial_{ef}g_{bc}g_{ad}g^{ef} + 4\partial_{be}g_{cf}g_{ad}g^{ef}\\
&\quad + 2\partial_b g_{ef}\partial_c g_{gh}g_{ad}g^{eg}g^{fh} + 4\partial_e g_{bf}\partial_g g_{ch}g_{ad}g^{eg}g^{fh} - 4\partial_e g_{bf}\partial_g g_{ch}g_{ad}g^{eh}g^{fg}\\
&\quad - 4\partial_b g_{ce}\partial_f g_{gh}g_{ad}g^{eg}g^{fh} + 2\partial_b g_{ce}\partial_f g_{gh}g_{ad}g^{ef}g^{gh} - 4\partial_c g_{be}\partial_f g_{gh}g_{ad}g^{eg}g^{fh}\\
&\quad + 2\partial_c g_{be}\partial_f g_{gh}g_{ad}g^{ef}g^{gh} + 4\partial_e g_{bc}\partial_f g_{gh}g_{ad}g^{eg}g^{fh} - 2\partial_e g_{bc}\partial_f g_{gh}g_{ad}g^{ef}g^{gh}\\
&\quad + 4\partial_{ef}g_{gh}g_{ac}g_{bd}g^{eg}g^{fh} - 4\partial_{ef}g_{gh}g_{ad}g_{bc}g^{eg}g^{fh} - 4\partial_{ef}g_{gh}g_{ac}g_{bd}g^{ef}g^{gh}\\
&\quad + 4\partial_{ef}g_{gh}g_{ad}g_{bc}g^{ef}g^{gh} - 2\partial_e g_{fg}\partial_h g_{ij}g_{ac}g_{bd}g^{ei}g^{fh}g^{gj} + 2\partial_e g_{fg}\partial_h g_{ij}g_{ad}g_{bc}g^{ei}g^{fh}g^{gj}\\
&\quad + 3\partial_e g_{fg}\partial_h g_{ij}g_{ac}g_{bd}g^{eh}g^{fi}g^{gj} - 3\partial_e g_{fg}\partial_h g_{ij}g_{ad}g_{bc}g^{eh}g^{fi}g^{gj} - 4\partial_e g_{fg}\partial_h g_{ij}g_{ac}g_{bd}g^{ef}g^{gi}g^{hj}\\
&\quad + 4\partial_e g_{fg}\partial_h g_{ij}g_{ad}g_{bc}g^{ef}g^{gi}g^{hj} + 4\partial_e g_{fg}\partial_h g_{ij}g_{ac}g_{bd}g^{ef}g^{gh}g^{ij} - 4\partial_e g_{fg}\partial_h g_{ij}g_{ad}g_{bc}g^{ef}g^{gh}g^{ij}\\
&\quad - \partial_e g_{fg}\partial_h g_{ij}g_{ac}g_{bd}g^{eh}g^{fg}g^{ij} + \partial_e g_{fg}\partial_h g_{ij}g_{ad}g_{bc}g^{eh}g^{fg}g^{ij} && /33/\\
&= 0 && /39/
\end{aligned}
$$

13.2 Proof using an orthonormal basis

This method is entirely local, that is, it only requires values of the metric and Riemann tensors at some arbitrarily chosen point. In contrast, the previous method required knowledge of the metric in a neighbourhood of a point (in order to compute its various derivatives).

One of the keys steps in this method is to use the basic definitions $R_{ab} = g^{cd}R_{acbd}$ and $R = g^{ab}R_{ab}$ to express the Weyl tensor entirely in terms of R_{abcd}, g_{ab} and g^{ab}. This leads to

$$
\begin{aligned}
C_{abcd} = R_{abcd} &- g^{ef}R_{aecf}g_{bd} + g^{fe}R_{afde}g_{bc} - g_{ac}g^{fe}R_{bfde}\\
&+ g_{ad}g^{ef}R_{becf} + \frac{1}{2}g^{ef}g^{gh}R_{egfh}\left(g_{ac}g_{bd} - g_{ad}g_{bc}\right)
\end{aligned}
$$

Consider now three vectors e^a_i, $i = x, y, z$, that form an orthonormal basis at the chosen point. Then the metric g_{ab} and its inverse g^{ab} can be written as

$$g_{ab} = e^x_a e^x_b + e^y_a e^y_b + e^z_a e^z_b \tag{13.1}$$

$$g^{ab} = e^a_x e^b_x + e^a_y e^b_y + e^a_z e^b_z \tag{13.2}$$

where e^i_a are dual to e^a_i, that is

$$e^a_i e^j_a = \delta_i{}^j \tag{13.3}$$
$$e^a_i e^i_b = \delta^a{}_b \tag{13.4}$$

The main part of the calculation is to show that the frame components

$$\hat{C}_{ijkl} = C_{abde} e^a_i e^b_j e^c_k e^d_l \tag{13.5}$$

of the Weyl tensor vanish (and hence, using (13.4), that the Weyl tensor must also vanish). It is sufficient to compute just two frame components, $\hat{C}_{xyxy}$ and $\hat{C}_{xyxz}$ as all other frame components can be found by simply permuting x, y and z.

The scene is now set for Cadabra. The key elements in the Cadabra code include a declaration of a Riemann tensor

```
R_{a b c d}::RiemannTensor.
```

together with rules for the Weyl tensor and friends

```
Rab     := R_{a b} -> g^{c d} R_{a c b d}.
Rscalar := R -> g^{a b} R_{a b}.

# Weyl tensor in 3-dimensions

Cabcd := R_{a b c d} - (R_{a c} g_{b d} - R_{a d} g_{b c})
                     - (g_{a c} R_{b d} - g_{a d} R_{b c})
               + 1/2 R (g_{a c} g_{b d} - g_{a d} g_{b c}).

substitute (Cabcd, Rscalar)
substitute (Cabcd, Rab)

Cabcd := C_{a b c d} -> @(Cabcd).
```

and rules that define the metric inverse and the orthonormal basis

```
gab := g^{a b} -> ex^{a} ex^{b} + ey^{a} ey^{b} + ez^{a} ez^{b}.

ortho := {ex^{a} ex^{b} g_{a b} -> 1,
          ey^{a} ey^{b} g_{a b} -> 1,
          ez^{a} ez^{b} g_{a b} -> 1,
          ex^{a} ey^{b} g_{a b} -> 0, ex^{a} ez^{b} g_{a b} -> 0,
          ey^{a} ex^{b} g_{a b} -> 0, ey^{a} ez^{b} g_{a b} -> 0,
          ez^{a} ex^{b} g_{a b} -> 0, ez^{a} ey^{b} g_{a b} -> 0}.
```

The last part of the calculation is to apply these rules to the frame components $\hat{C}_{xyxy}$ and $\hat{C}_{xyxz}$. For $\hat{C}_{xyxy}$ the code is

```
Cxyxy := C_{a b c d} ex^{a} ey^{b} ex^{c} ey^{d}.

substitute     (Cxyxy,Cabcd)
distribute     (Cxyxy)
substitute     (Cxyxy, ortho, repeat=True)
substitute     (Cxyxy, gab)
```

```
7   distribute      (Cxyxy)
8
9   sort_product    (Cxyxy)
10  rename_dummies  (Cxyxy)
11  canonicalise    (Cxyxy)
```

which leads to the following output

$$
\begin{aligned}
C_{abcd}e_x^a e_y^b e_x^c e_y^d &= \Big(R_{abcd} - g^{ef}R_{aecf}g_{bd} + g^{fe}R_{afde}g_{bc} - g_{ac}g^{fe}R_{bfde} + g_{ad}g^{ef}R_{becf} \\
&\qquad + \frac{1}{2}g^{ef}g^{gh}R_{egfh}g_{ac}g_{bd} - \frac{1}{2}g^{ef}g^{gh}R_{egfh}g_{ad}g_{bc}\Big)\, e_x^a e_y^b e_x^c e_y^d && /3/ \\
&= R_{abcd}e_x^a e_y^b e_x^c e_y^d - g^{ef}R_{aecf}g_{bd}e_x^a e_y^b e_x^c e_y^d + g^{fe}R_{afde}g_{bc}e_x^a e_y^b e_x^c e_y^d \\
&\quad - g_{ac}g^{fe}R_{bfde}e_x^a e_y^b e_x^c e_y^d + g_{ad}g^{ef}R_{becf}e_x^a e_y^b e_x^c e_y^d + \frac{1}{2}g^{ef}g^{gh}R_{egfh}g_{ac}g_{bd}e_x^a e_y^b e_x^c e_y^d \\
&\quad - \frac{1}{2}g^{ef}g^{gh}R_{egfh}g_{ad}g_{bc}e_x^a e_y^b e_x^c e_y^d && /4/ \\
&= R_{abcd}e_x^a e_y^b e_x^c e_y^d - g^{ef}R_{aecf}e_x^a e_x^c - g^{fe}R_{bfde}e_y^b e_y^d + \frac{1}{2}g^{ef}g^{gh}R_{egfh} && /5/ \\
&= R_{abcd}e_x^a e_y^b e_x^c e_y^d - \left(e_x^e e_x^f + e_y^e e_y^f + e_z^e e_z^f\right) R_{aecf}e_x^a e_x^c - \left(e_x^f e_x^e + e_y^f e_y^e + e_z^f e_z^e\right) R_{bfde}e_y^b e_y^d \\
&\quad + \frac{1}{2}\left(e_x^e e_x^f + e_y^e e_y^f + e_z^e e_z^f\right)\left(e_x^g e_x^h + e_y^g e_y^h + e_z^g e_z^h\right) R_{egfh} && /6/ \\
&= \frac{1}{2}R_{abcd}e_x^a e_x^c e_y^b e_y^d - \frac{1}{2}R_{abcd}e_x^a e_x^c e_x^b e_x^d - \frac{1}{2}R_{abcd}e_x^a e_x^c e_z^b e_z^d - R_{abcd}e_x^d e_x^b e_y^a e_y^c \\
&\quad - R_{abcd}e_y^a e_y^c e_y^d e_y^b - R_{abcd}e_y^a e_y^c e_z^d e_z^b + \frac{1}{2}R_{abcd}e_x^b e_x^d e_y^a e_y^c + \frac{1}{2}R_{abcd}e_y^a e_y^c e_y^b e_y^d \\
&\quad + \frac{1}{2}R_{abcd}e_y^a e_y^c e_z^b e_z^d + \frac{1}{2}R_{abcd}e_x^b e_x^d e_z^a e_z^c + \frac{1}{2}R_{abcd}e_y^b e_y^d e_z^a e_z^c + \frac{1}{2}R_{abcd}e_z^a e_z^c e_z^b e_z^d && /10/ \\
&= 0 && /11/
\end{aligned}
$$

showing clearly (in the last line) that $\hat{C}_{xyxz} = 0$. Similar code can be used to show that $\hat{C}_{xyxz} = 0$.

14 Conformal invariance of the Weyl tensor

The Weyl tensor has the property that it is conformally invariant under conformal transformations of the metric. That is, if two metrics, g and $\overline{g}$, are related by a conformal transformation, $\overline{g} = \phi g$ for some scalar function ϕ, then their corresponding Weyl tensors are equal, $\overline{C}^a{}_{bcd} = C^a{}_{bcd}$. This simple result can be shown by direct computation. Though this result is true in any number of dimensions the specific case of four dimensions (in this example) is sufficient to demonstrate the ideas behind a general proof. Note that the following computation is based on the related expression $\overline{C}_{abcd} = \phi C_{abcd}$.

The computation is similar to the previous example and begins by first computing a general expression for C_{abcd} by forming an appropriate combination of rules. Then a copy of the result is made (this is the Weyl tensor on the base metric g),

```
baseC := @(Cabcd).
```

followed by a rule defining the conformal transformation

```
conformal := {g_{a b} -> \phi g_{a b}, g^{a b} -> (1/phi) g^{a b}}.
```

The rule is applied to the current version of `C_{a b c d}` followed by some basic housekeeping

```
substitute   (Cabcd, conformal)
product_rule (Cabcd)
distribute   (Cabcd)
product_rule (Cabcd)
distribute   (Cabcd)
map_sympy    (Cabcd, "simplify")
```

Note that two rounds of the product rule are required because the conformal factor is buried inside a set of second order partial derivatives. The result is the Weyl tensor for the conformal metric. This is copied to a new object and then the difference between the two Weyl tensors is computed

```
confC := @(Cabcd).
diff  := @(confC) - \phi @(baseC).
```

The game now is to show that `diff` is zero. The code then make uses of the basic identities (in 4 dimensions)

$$g_{ab}g^{ab} = 4, \quad g_{ac}g^{cb} = \delta^b{}_a, \quad \delta^a{}_a = 4$$

to simplify the expression `diff`. The result is zero (as expected).

15 The BSSN equations

Einstein's equation of General Relativity are most simply described in the full 4-dimensional form as

$$R_{ab} - \frac{1}{2} g_{ab} R = \kappa T_{ab} \tag{15.1}$$

They can also be recast in the form of a Cauchy initial value problem in which a 3-dimensional metric is evolved forward in time from a given set of initial conditions. One such formulation is due to Arnowitt, Deser and Misner, widely known as the ADM 3+1 formulation. In one of its simplest forms[d], the ADM 3+1 evolution equations can be written as

$$\frac{\partial g_{ij}}{\partial t} = -2NK_{ij} \tag{15.2}$$

$$\frac{\partial K_{ij}}{\partial t} = -D_{ij}N + N(R_{ij} + KK_{ij} - 2K_{im}K_{jn}g^{mn}) \tag{15.3}$$

where g_{ij} is the 3-metric, K_{ij} is the extrinsic curvature, R_{ij} is the Ricci tensor, D is the metric compatible covariant derivative and finally, N is the lapse function (which can be freely specified though subject to $N > 0$).

For many years the ADM 3+1 equations were the cornerstone of computational general relativity. Unfortunately they proved to be less than ideal for long term evolutions of one or more black holes (the evolutions were highly unstable). In recent times an alternative set of equations, first proposed by Shibata and Nakamura [18] and later popularised by Baumgarte and Shapiro [19], now known as the BSSN equations, have come to dominate the field (as they do allow stable long term evolutions of black hole systems).

The BSSN evolution equations, for vacuum spacetimes and a zero shift vector, are given by

$$\frac{\partial \phi}{\partial t} = -\frac{1}{6} NK \tag{15.4}$$

$$\frac{\partial \bar{g}_{ij}}{\partial t} = -2N\bar{A}_{ij} \tag{15.5}$$

$$\frac{\partial K}{\partial t} = -g^{ij} D_{ij} N + N(\bar{A}_{ij}\bar{A}^{ij} + \frac{1}{3}K^2) \tag{15.6}$$

$$\frac{\partial \bar{A}_{ij}}{\partial t} = N(K\bar{A}_{ij} - 2\bar{A}_{ik}\bar{A}^k{}_j) + \exp(-4\phi)(NR_{ij} - D_{ij}N - \frac{1}{3} g_{ij}(NR_{kl} - D_{kl}N)g^{kl}) \tag{15.7}$$

$$\frac{\partial \bar{\Gamma}^i}{\partial t} = -2\partial_j \left(N\bar{A}^{ij}\right) \tag{15.8}$$

$$= -2\bar{A}^{ij}\partial_j N + 2N(\bar{\Gamma}^i_{jk}\bar{A}^{kj} - \frac{2}{3}\bar{g}^{ij}\partial_j K + 6\bar{A}^{ij}\partial_j\phi) \tag{15.9}$$

The Ricci tensor could be computed directly from the 3-metric but part of the black magic of the BSSN formulation is to use

$$\begin{aligned} R_{ij} = {} & -2\bar{D}_{ij}\phi - 2\bar{g}_{ij}\bar{g}^{mn}\bar{D}_{mn}\phi + 4\bar{D}_i\phi\bar{D}_j\phi - 4\bar{g}_{ij}\bar{g}^{mn}\bar{D}_m\phi\bar{D}_n\phi \\ & - \frac{1}{2}\bar{g}^{lm}\partial_{lm}\bar{g}_{ij} + \bar{g}_{k(i}\partial_{j)}\bar{\Gamma}^k + \bar{\Gamma}^k\bar{\Gamma}_{(ij)k} + \bar{g}^{lm}\bar{g}^{kp}(\bar{\Gamma}_{pl(i}\bar{\Gamma}_{j)km} + \bar{\Gamma}_{kim}\bar{\Gamma}_{plj}) \end{aligned} \tag{15.10}$$

[d]That is, for vacuum spacetimes using coordinates with a zero shift vector.

The dynamical variables of the ADM formulation, g_{ij}, K_{ij} are related to the those of the BSSN formulation, $K, \phi, \bar{g}_{ij}, \bar{A}_{ij}, \bar{\Gamma}^i$ by the equations

$$\begin{aligned} K &= g^{ij}K_{ij} && (15.11)\\ e^{4\phi} &= g^{1/3} = (\det(g_{ij}))^{1/3} && (15.12)\\ \bar{g}_{ij} &= e^{-4\phi}g_{ij} && (15.13)\\ \bar{A}_{ij} &= e^{-4\phi}\left(K_{ij} - \frac{1}{3}g_{ij}K\right) && (15.14)\\ \bar{\Gamma}^i &= \bar{g}^{jk}\bar{\Gamma}^i_{jk} = -\bar{g}^{ij}{}_{,j} && (15.15)\end{aligned}$$

These equations can be used to derive the BSSN equations from the ADM equations.

After a drawn out preamble here is the point of this example – to show how Cadabra can be used to derive the BSSN equations from the ADM equations. The calculations are non-trivial so only the first two of the five evolution equations will be derived here. The derivation of the full set of equations (including the constraint equations) can be found on the website https://github.com/leo-brewin/adm-bssn-equations

Check: Do I derive all the constraint equations? No, not as a single file but I have used versions of the Hamiltonian in eqtns11.tex and eqtns12.tex. I will create a new file just for the Hamiltonian.

It should be noted that the equations given above (15.4-15.15) are a subset of the full set of BSSN equations. For full details see [20, 21]

15.1 Evolution equation for ϕ

The key elements in the Cadabra code for the first BSSN equation (15.4) are the four rules

```
phi      := \phi                       -> (1/12) \log(detg).
gdotK    := g^{i j} K_{i j}            -> trK.
DdetgDt  := \partial_{t}{detg}         -> detg g^{i j} \partial_{t}{g_{i j}}.
DgijDt   := \partial_{t}{g_{i j}}      -> -2 N K_{i j}.
```

The first three rules follow from the definitions of ϕ, $\det g$ and K while the final rule is the original ADM equation for $\partial g_{ij}/\partial t$. Two other rules are also included as they help train Cadabra to do basic calculus

```
dlog     := \partial_{a?}{\log(A?)} -> (1/A?)\partial_{a?}{A?}.
dexp     := \partial_{a?}{\exp(A?)} -> \exp(A?)\partial_{a?}{A?}.
```

The main body of the code begins with a single line

```
dotphi   := \partial_{t}{\phi}.
```

followed be a series of substitutions

```
substitute (dotphi, phi)
substitute (dotphi, dlog)
substitute (dotphi, DdetgDt)
substitute (dotphi, DgijDt)
substitute (dotphi, gdotK)
```

```
map_sympy  (dotphi, "simplify")
```

The step-by-step results are as follows

$$\begin{aligned}
\frac{\partial\phi}{\partial t} &= \frac{1}{12}\partial_t\left(\log\left(g\right)\right) && /2/\\
&= \frac{1}{12}g^{-1}\partial_t g && /3/\\
&= \frac{1}{12}g^{-1}gg^{ij}\partial_t g_{ij} && /4/\\
&= -\frac{1}{6}g^{-1}gg^{ij}NK_{ij} && /5/\\
&= -\frac{1}{6}g^{-1}gKN && /6/\\
&= -\frac{1}{6}KN && /7/
\end{aligned}$$

15.2 Evolution equation for $\bar{g}_{ij}$

A similar set of rules and substitutions can be used to obtain the second BSSN equation (15.5). In this case the essential rules are

```
DphiDt := \partial_{t}{\phi}   -> @(dotphi).
gBarij := gBar_{i j}            -> \exp(-4\phi) g_{i j}.
Kij    := K_{i j}               -> A_{i j} + (1/3) g_{i j} trK.
A2ABar := \exp(-4\phi) A_{i j} -> ABar_{i j}.
```

Note that the first rule is built using the result of the computation for $\partial\phi/\partial t$. This construction (of building rules to record key results) is used frequently in the full BSSN code[e]. It avoids having to copy-paste results for later use and thus also avoids any transcription errors. The other rules are again built directly from the basic definitions of the BSSN variables.

The starting point for the main calculation is

```
dotgBarij := \partial_{t}{gBar_{i j}}.
```

followed by some substitutions, a product rule and dab of housekeeping

```
substitute   (dotgBarij, gBarij)
product_rule (dotgBarij)
substitute   (dotgBarij, dexp)
substitute   (dotgBarij, DgijDt)
substitute   (dotgBarij, DphiDt)
substitute   (dotgBarij, Kij)
distribute   (dotgBarij)
map_sympy    (dotgBarij, "simplify")
substitute   (dotgBarij, A2ABar)
```

[e]On the website https://github.com/leo-brewin/adm-bssn-equations

The corresponding output is

$$\begin{aligned}
\frac{\partial \bar{g}_{ij}}{\partial t} &= \partial_t \left(\exp\left(-4\phi\right) g_{ij}\right) && /2/ \\
&= \partial_t \left(\exp\left(-4\phi\right)\right) g_{ij} + \exp\left(-4\phi\right) \partial_t g_{ij} && /3/ \\
&= -4\exp\left(-4\phi\right) \partial_t \phi g_{ij} + \exp\left(-4\phi\right) \partial_t g_{ij} && /4/ \\
&= -4\exp\left(-4\phi\right) \partial_t \phi g_{ij} - 2\exp\left(-4\phi\right) N K_{ij} && /5/ \\
&= \frac{2}{3}\exp\left(-4\phi\right) K N g_{ij} - 2\exp\left(-4\phi\right) N K_{ij} && /6/ \\
&= \frac{2}{3}\exp\left(-4\phi\right) K N g_{ij} - 2\exp\left(-4\phi\right) N \left(A_{ij} + \frac{1}{3} g_{ij} K\right) && /7/ \\
&= \frac{2}{3}\exp\left(-4\phi\right) K N g_{ij} - 2\exp\left(-4\phi\right) N A_{ij} - \frac{2}{3}\exp\left(-4\phi\right) N g_{ij} K && /8/ \\
&= -2N\exp\left(-4\phi\right) A_{ij} && /9/ \\
&= -2N\bar{A}_{ij} && /10/
\end{aligned}$$

15.3 A numerical code

The website https://github.com/leo-brewin/adm-bssn-equations contains all of the Cadabra code for a complete derivation the BSSN equations (with zero shift) from the ADM equations. A companion website, https://github.com/leo-brewin/adm-bssn-numerical-code, contains further Cadabra code that converts the BSSN equations into a working numerical code.

All of the tools in that second website are based on material already covered in this tutorial (in particular, Example 7 for exporting tensor expression as C-code).

Using a symbolic package (in this case Cadabra) to turn a set of partial differential equations (the BSSN equations) into a numerical code has great advantages. It frees the researcher from the tedium of writing extensive code (try writing a code for R_{ab} from the metric by hand), it minimises the risk of coding errors and it allows for much quicker development of new codes as changes are made in the underlying mathematics (e.g., shifting from the ADM to the BSSN equations). This approach is quite common in the computational general relativity community, see for example the papers by Husa etal. [10] and Ruchlin etal. [11].

Part 3 Common traps and errors.

Despite our best efforts, bugs do creep in from time to time. Often the errors are immediately obvious but on other occasions a great deal of head scratching and scouring of web pages fills in the time before the light-bulb moment arrives. Here are some examples of what can go wrong, how to spot the errors and tips on how to avoid them in the first place.

Problems with indices

If you have never encountered the Cadabra runtime error

```
RuntimeError: Free indices in different terms in a sum do not match.
```

consider yourself lucky. For those who have (author included) here are some examples demonstrating various ways to encounter this error.

1. **Inconsistent free indices**

   This is trivial – the free indices on `A` and `B` do not match.

   ```
   foo := A_{a} + B_{b};
   ```

2. **Incorrect number of free indices**

   Another trivial example. Maybe a typo (one too many indices) on `B_{a b}`?

   ```
   foo := A_{a} + B_{a b};
   ```

3. **Missing spaces**

   The intention in the following line is to create an expression with two free indices

   ```
   foo := A_{ab} + B_{a b};
   ```

   The problem here is that Cadabra will take `_{ab}` to be a single index. Always include a space between indices (unless the indices have a natural separator like the slash in LaTeX names, e.g., `_{\alpha\beta}` would be accepted as a pair of indices).

4. **Forgetting to declare the derivative operator**

   Here is a simple and apparently correct use of indices.

   ```
   foo := A_{a b} + \partial_{a}{A_{b}};
   ```

   So why would Cadabra complain? The answer is that by forgetting to declare `\partial` as a derivative operator, Cadabra will interpret `\partial_{a}{A_{b}}` as a function call with argument `A_{b}`. Thus it thinks that this term has just one free index, namely, `_{a}`. Hence the error.

5. **Cavalier use of `@(...)`**

   The `@(...)` construct is extremely useful but it also requires the user to take great care less the dreaded index problem pops up. Here is a simple example

   ```
   foo := A_{a};
   bah := B_{b};
   meh := @(A) + @(B);
   ```

   The problem here is obvious – the free indices on `@(A)` and `@(B)` clearly do not match. Though this error is startlingly obvious in this example it may be much harder to detect in codes where the computation of `A` and `B` are buried deep in parts of the code far removed from each other and their use in `@(A)+@(B)`. One way to avoid this problem is to use rules that define `A` and `B`. Here is a short example.

```
foo := A_{a};
ruleA := TmpA_{a} -> @(foo);
...
bah := B_{b};
ruleB := TmpB_{b} -> @(bah);
...
meh := TmpA_{c} + TmpB_{c};
substitute (meh, ruleA+ruleB)
```

The ellipses in the above denote some intervening code. The rules are created as soon as the expressions `foo` and `bah` have been created. In most cases the right hand side of `foo` and `bah` will be substantially more complicated than that given above.

6. **Upstairs/downstairs index clash**

   The free indices in an expression must be consistent with regards to being upstairs or downstairs when using `Indices(position=independent)`. Thus the following code snippet will cause grief for Cadabra.

```
foo := A_{a} + B^{a}.
```

Problems with derivatives

6. **Forgetting to declare the derivative operator**

   This has already been noted (see see item 4 above) – but a reminder can not hurt.

7. **Forgetting to enclose the derivative argument in $\{\ldots\}$**

   The printed output for the following

```
foo := A_{a b} + \partial_{a} A_{b};
```

will look like

$$A_{ab} + \partial_a A_b$$

which seems fine. But without the `{...}` enclosing the `A_{b}` term, Cadabra will interpret `\partial_{a} A_{b}` as a product of `\partial_{a}` with `A_{b}`. You can see that this is so by calling `sort_product` on `foo`. The output will be

$$A_{ab} + A_b \partial_a$$

8. **Avoid applying `canonicalise` to partial derivatives**

   Calling `canonicalise` on an expression like

```
foo := A_{a} \partial_{b}{B^{a}};
```

might result in index raising/lowering of the dummy index `a`. In general relativity this would not be allowed (except for the trivial case where the metric components are constants). One way to avoid this problem is to use either `Indices(position=fixed)` or `Indices(position=independent)`. This will force `canonicalise` to leave the indices as is. Another option (if possible) is to only use metric compatible derivative operators.

Problems with substitution rules

9. Rules using `$...$` must be confined to a single line

Rules built using `$...$` must be defined on a single line. The following example

```
substitute (foo, $A_{a}->B_{a},
                 C_{a}->D_{a}$);
```

will raise a syntax error

```
SyntaxError: invalid syntax.
```

You can fix this either by collapsing the `$...$` to a single line or by creating a named substitution rule (these can be split over multiple lines) as in the following code

```
myRule := {A_{a}->B_{a},
           C_{a}->D_{a}}.
substitute (foo, myRule);
```

10. Only use `->` to change index structure

There are occasions where indices need to added or deleted from expressions. Doing so using an equality rule like `A_{a b} = A_{a}` will raise a runtime error. For example, the following code

```
foo := A_{a b};
substitute (foo, $A_{a b} = A_{a}$)
```

causes Cadabra great grief, reporting that

```
RuntimeError: Free indices on lhs and rhs do not match.
```

The correct code is

```
foo := A_{a b};
substitute (foo, $A_{a b} -> A_{a}$);
```

with output A_a as expected.

11. Use care when using `->` to change index structure

Changing the index structure of an expression can cause runtime errors. Here is a simple example.

```
foo := A_{a} x^{a} + B_{b} x^{b}.
substitute (foo, $x^{a} -> 1$)
```

The problem here is that the result for `foo` is `A_{a} + B_{b}` and though Cadabra does not report an index mis-match error at this point it will do so later (most likely at the point when `foo` is coupled to some other expression). The solution to this problem is to ensure that the each term in the expression uses the same index on `x`. Here is a corrected version of the code.

```
foo := A_{a} x^{a} + B_{b} x^{b}.
rename_dummies (foo)
```

```
substitute     (foo, $x^{a} -> 1$);
```

This works because the result after renaming the dummy indices is

```
foo := A_{a} x^{a} + B_{a} x^{a}
```

But had the initial expression for `foo` been

```
foo := A_{b} B^{b} C_{a} x^{a} + D_{d} x^{d};
```

then this simple trick of renaming the dummies would not be sufficient to avoid the later problem when applying `x^{a} -> 1`. In this case the call to `rename_dummies` will return

```
foo := A_{a} B^{a} C_{b} x^{b} + D_{a} x^{a};
```

The problem here is that once again the `x` terms do not share a common index. This occurs because the renaming of dummy indices occurs left to right. As the first term requires two dummy indices while the second requires one the first `x` will be given a different dummy to that assigned to the second `x`.

This minor problem can solved by first using `sort_product` to bring the `x` factors to the left of all other terms. Here is a code that does the job.

```
{x^{a},A_{a},B^{a},D_{a}}::SortOrder.
foo := A_{a} B^{a} C_{b} x^{b} + D_{a} x^{a};
sort_product   (foo)
rename_dummies (foo)
canonicalise   (foo)
substitute     (foo, $x^{a} -> 1$);
```

The corresponding output is

$$A_b B^b C_a + D_a$$

This is one of the reasons why numerous exercises on sorting were included in the collection at the end of Example 1. Note that in this case `rename_dummies` did not align the `x` indices. That job fell to `canonicalise`. The combination of `sort_product`, `rename_dummies` and `canoniclaise` appears throughout this tutorial in the examples and exercises. It is a very standard combination.

The take home point here is that careful inspection of the expression is required before operations that alter the index structure are applied.

Miscellaneous

12. Syntax error

This covers a whole raft of errors and in most cases the fix will be obvious. Here are a few things to look for.

- Check the termination character – a dot, a semi-colon or the closing parenthesis of a function call. The dot and semi-colon are only used on Cadabra statements.
- Check the assignment operator, use `:=` for Cadabra and `=` for Python.

- Do not use underscores in symbol names.
- Use standard LaTeX names such as `\alpha,\beta,\mu` etc.
- Do not use `return @(foo)`. The correct return is `return foo`.

13. Problems with LaTeXForm

If you want to specify the LaTeX form for an object that carries indices you must use `{#}`, otherwise do not use `{#}`. Here are two simple example.

```
foo{#}::LaTeXForm{"{\bar \alpha}"}.    # matches objects foo with indices
bah::LaTeXForm{"{\hat \beta}"}.        # matches objects bah without indices
```

14. Correct form of exponential function

Be aware that Cadabra will treat `e^{a}` as a tensor with one upstairs index. If you wanted the exponential function then you should write `\exp{a}` or `e**{a}`.

15. Do not use underscore in expression names

This has been mentioned before – underscores denote subscripts and thus should not be used as part of an expression name (though their use in function names is perfectly okay).

16. Horizontal alignment of indices

Using braces `{}` around indices, even single indices, ensures that the printed version of the tensor will have its indices in sequential columns. Thus `R^{a}_{b c d}` will be printed as $R^{a}{}_{bcd}$ while `R^a_{b c d}` will be printed as R^a_{bcd}.

17. Excessively long lines

Each statement in the following fragment will raise a Cadabra syntax error.

```
{\alpha,\beta,\gamma,\delta,
 \mu,\nu,\sigma,\rho,\tau,\theta}::Indices.

{R_{\alpha\beta\gamma\delta},
 \partial_{\mu}{R_{\alpha\beta\gamma\delta}}}::SortOrder.

 substitute (foo, $R -> R_{\mu\nu} g^{\mu\nu},
                  R_{\mu\nu} -> R_{\alpha\mu\beta\nu} g^{\alpha\beta}$)
```

One solution is to condense each statement to a single line (one for each statement). That will work but may lead to excessively long lines. There is an alternative – convert the (single-line) statements into pure Python (using the command line tool `cadabra2python`) and then add suitable line breaks. Suppose that the (single-line) statements are in the file `foo.cdb`. You can then create the Python equivalent `foo.py` using (on the command line)

```
cadabra2python foo.cdb foo.py
```

The file `foo.py` will contain the following lines

```
__cdbtmp__ = Indices(Ex(r'''{\alpha,\beta,\gamma,\delta,\mu,\nu,\sigma,\rho,\tau,\theta

__cdbtmp__ = SortOrder(Ex(r'''{R_{\alpha\beta\gamma\delta},\partial_{\mu}{R_{\alpha\bet

 substitute (foo, Ex(r'''R -> R_{\mu\nu} g^{\mu\nu},R_{\mu\nu} -> R_{\alpha\mu\beta\nu}
```

This does not seem like much of an improvement but the good news is that as most of the text is written as strings the issue of line breaking is now trivial – strings are easily split across lines. This is also a good time to do a bit tidying up (replacing triple quotes with single quotes and `__cdbtmp__` with `tmp`). The tidied version of `foo.py` is now

```
tmp = Indices(Ex(r'{\alpha,\beta,\gamma,\delta,'+
                 r'\mu,\nu,\sigma,\rho,\tau,\theta}'), Ex(r'') )

tmp = SortOrder(Ex(r'{R_{\alpha\beta\gamma\delta},'+
                   r'\partial_{\mu}{R_{\alpha\beta\gamma\delta}}}'), Ex(r'') )

substitute (foo, Ex(r'R -> R_{\mu\nu} g^{\mu\nu},'+
                    r'R_{\mu\nu} -> R_{\alpha\mu\beta\nu} g^{\alpha\beta}', False))
```

This code fragment will be happily accepted by Cadabra (though it is a matter of opinion whether the aesthetics of this version are an improvement over the original single-line statements).

After running a few experiments you should be able to infer the basic actions of `cadabra2python`. For a typical property like

```
{list of things}::PropertyName(arguments).
```

the conversion will produce (after a bit of tidying up)

```
tmp = PropertyName ( Ex(r'list of things'), Ex(r'arguments') )
```

while for a typical algorithm like

```
Algorithm (foo, $a substitution rule$)
```

the result will be (again after manual tidying up)

```
Algorithm (foo, Ex(r'a substitution rule', False))
```

Note that the conversion can introduce some odd artefacts. For example, the following fragment

```
{a,b,c}::Indices(position=indpendent).

substitute (foo, $A->B$)
```

will be converted as

```
tmp = Indices(Ex(r'{a,b,c}'), Ex(r'position=independent)') )

substitute (foo, Ex(r'''A->B''', False))
```

The stray closing parenthesis in `Ex(r'position=indpendent)')` and the `False` argument in the `substitute (foo,Ex(...))` call both appear to serve no purpose and can be deleted.

Is the above statement correct?

Part 4 Further reading.

Webpages

https://github.com/kpeeters/cadabra2.git
This is the GitHub repository for Cadabra2. You can clone the site using

```
git clone https://github.com/kpeeters/cadabra2.git
```

This will create a `cadabra2` directory containing the complete source code. It also contains full instructions on how to compile and install the code (see `cadabra2/README.rst`).

https://cadabra.science/qa/questions
This is a popular site for posting Cadabra questions and answers. Anyone can read the questions and answers but to post to the site you will need to register.

https://cadabra.science/man.html
This site describes every property and algorithm supported by Cadabra. It is the first place to go when looking for information about a property or an algorithm.

https://cadabra.science/help.html
This is the main online reference for Cadabra. It is written as series of short tutorials each covering a key aspect of Cadabra. Topics covered include basic syntax for writing expressions, properties and algorithms, basic input/output, how to manipulate expressions and an introduction to programming in Cadabra.

https://cadabra.science/notebooks/ref_patterns.html
This topic in the reference guide provides full details on how use the `?` and `??` patterns. It also discusses more powerful pattern matching using conditional patterns and regular expressions (neither of which are described in this tutorial).

https://cadabra.science/notebooks/ref_programming.html
This topic contains a good discussion on how expressions are stored in Cadabra. It also describes how you can access and manipulate the elements of an expression (such as its indices and the individual terms). The topic contains a nice function that will return the covariant derivative for *any* tensor.

https://cadabra.science/tutorials.html
This is a collection of tutorials showcasing the main features of Cadabra. The tutorials can be viewed online or they can be downloaded as Cadabra notebooks (and thus allowing experiments to be run in the Cadabra gui).

https://cadabra.science/user_notebooks.html
This is set of user contributed notebooks. One notebook (by Mattia Scomparin) shows how Cadabra can be used to derive the non-vacuum Einstein equations from Hilbert action integral. Another notebook (by Oscar Castillo-Felisola) uses differential forms to derive the second Bianchi identities.

https://github.com/leo-brewin/cadabra-tutorial
This is the GitHub repository for this tutorial. It contains all of the Cadabra and LaTeX sources as well the pdf files generated from those sources. The source files are written in a hybrid syntax that has the Cadabra code embedded within the LaTeX source. These hybrid sources can be compiled using a small set of tools (Python scripts and LaTeX style files) all of which can be obtained from the authors GitHub site (i.e., the very next web page).

https://github.com/leo-brewin/hybrid-latex
This site contains all the tools needed to process the hybrid LaTeX/Cadabra files used int his

tutorial. It also contains similar tools for LaTeX sources with embedded Maple, Mathematica, Matlab and Python code. Some people may find these tools useful beyond their use in this tutorial.

https://github.com/leo-brewin/riemann-normal-coords
This site contains all of the Cadabra code used to the authors' paper on Riemann Normal Coordinates [1].

https://github.com/leo-brewin/adm-bssn-equations
This site contains the full derivation of the BSSN equations from the ADM equations (for a zero shift vector). This extends the limited discussion given in Example 15 (where only two equations were derived).

https://github.com/leo-brewin/adm-bssn-numerical-code
This site contains a full 3+1 evolution code (written in Ada) for a Kasner T^3 cosmology. This includes the Cadabra code used to convert the BSSN equations into computer code suitable for use in the numerical integrators.

https://docs.python.org/3/
https://docs.python.org/3/reference/index.html
https://docs.python.org/3/tutorial/index.html#tutorial-index
The are the official sites for Python. They provide excellent information on all matters Python.

https://www.sympy.org/en/index.html
https://docs.sympy.org/latest/tutorial/index.html
These are excellent resources for learning how to use SymPy.

Notebooks

The Cadabra2 source code includes a wealth of sample notebook in the directory `cadabra2/examples`.

References

[1] Leo Brewin. "A brief introduction to Cadabra: a tool for tensor computations in General Relativity". In: *Computer Physics Communications* 181 (2010), pp. 489–498. eprint: arXiv:0903.2085.

[2] Peter Musgrave, Denis Pollney, and Kyall Lake. *GRTensorIII home page*. 2017. URL: https://github.com/grtensor/grtensor.

[3] Steven M Christensen and Leonard Parker. *MathTensor home page*. 2008. URL: http://smc.vnet.net/MathTensor.html.

[4] L. R. U. Manssur and R. Portugal. "The Canon package: a fast kernel for tensor manipulators". In: *Computer Physics Communications* 157.2 (2004). New version of Canon now found on Invar web page, as given above, pp. 173–180. DOI: 10.1016/S0010-4655(03)00494-6. URL: http://www.lncc.br/~portugal/Invar.html.

[5] R. Portugal and S. L. Sautú. "Applications of Maple to general relativity". In: *Computer Physics Communications* 105.2-3 (1997), pp. 233–253. DOI: 10.1016/S0010-4655(97)00078-7.

[6] José Martín-García. *xAct: Efficient tensor computer algebra for Mathematica*. 2008. URL: http://metric.iem.csic.es/Martin-Garcia/xAct/.

[7] Malcolm A. H. MacCallum. “Computer algebra in gravity research”. In: *Living Reviews in Relativity* 21 (2018). ISSN: 1433-8351. DOI: 10.1007/s41114-018-0015-6. URL: https://doi.org/10.1007/s41114-018-0015-6.

[8] Kasper Peeters. *The Cadabra2 GitHub reporistory*. 2017. URL: https://github.com/kpeeters/cadabra2.

[9] Kasper Peeters. *The Cadabra v2.x home page*. 2017. URL: https://cadabra.science/index.html.

[10] Sascha Husa, Ian Hinder, and Christiane Lechner. “Kranc: a Mathematica package to generate numerical codes for tensorial evolution equations”. In: *Computer Physics Communications* 174.12 (2006), pp. 983–1004. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2006.02.002. eprint: arXiv:gr-qc/0404023. URL: http://www.sciencedirect.com/science/article/pii/S0010465506001020.

[11] Ian Ruchlin, Zachariah B. Etienne, and Thomas W. Baumgarte. “SENR/NRPy+: Numerical relativity in singular curvilinear coordinate systems”. In: *Phys. Rev. D* 97 (6 2018), p. 064036. DOI: 10.1103/PhysRevD.97.064036. eprint: arXiv:1712.07658. URL: https://link.aps.org/doi/10.1103/PhysRevD.97.064036.

[12] Isaac Chavel. *Riemannian Geometry. A modern introduction, 2nd ed.* Cambridge University Press, Cambridge., 2006.

[13] S.S. Chern, W.H. Chen, and K.S. Lam. *Lectures on Differential Geometry*. World Scientific, Singapore, 2000.

[14] Luther Pfahler Eisenhart. *Riemannian Geometry*. Princeton University Press, Princeton, 1926.

[15] Alfred Gray. “The volume of a small geodesic ball of a Riemannian manifold”. In: *Michigan.Math.J.* 20 (1973), pp. 329–344.

[16] T.J. Willmore. *Riemannian Geometry*. Oxford University Press, Oxford, 1996.

[17] E. Calzetta, S. Habib, and B.L. Hu. “Quantum kinetic field theory in curved spacetime: Covariant Wigner function and Liouville-Vlasov equations”. In: *Phys. Rev. D* 37.10 (1988), pp. 2901–2919.

[18] Masaru Shibata and Takashi Nakamura. “Evolution of three-dimensional gravitational waves: Harmonic slicing case”. In: *Phys. Rev. D* 52 (1995), pp. 5428–5444.

[19] Thomas W. Baumgarte and Stuart L. Shapiro. “Numerical integration of Einstein’s field equations”. In: *Phys. Rev. D* 59 (1998), p. 024007. eprint: gr-qc/9810065.

[20] Miguel Alcubierre et al. “Towards a stable numerical evolution of strongly gravitating systems in general relativity: The conformal treatments”. In: *Phys. Rev. D* 62 (2000), p. 044034.

[21] Miguel Alcubierre et al. “Gauge conditions for long-term numerical black hole evolutions without excision”. In: *Phys. Rev. D* 67 (2003), p. 084023. eprint: gr-qc/0206072.