Preprint

A brief introduction to Cadabra: a tool for tensor computations in General Relativity

Leo Brewin

School of Mathematical Sciences
Monash University, 3800
Australia

10-Mar-2009

Abstract

Cadabra is a powerful computer program for the manipulation of tensor equations. It was designed for use in high energy physics but its rich structure and ease of use lends itself well to the routine computations required in General Relativity. Here we will present a series of simple examples showing how Cadabra may be used, including verifying that the Levi-Civita connection is a metric connection and a derivation of the Gauss equation between induced and ambient curvatures.

1 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 might we use 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, GRTensorII [1]). However, for the third example, the vast majority of researchers in General Relativity would do the calculations by hand. Recently a computer program, Cadabra [2, 3, 4], has appeared which may greatly assist us in these tedious tensor computations. This article will provide a brief introduction to Cadabra and how it can be used in General Relativity.

There are a number of programs that, to varying degrees, can manipulate tensor expressions, including GRTensorII [1], MathTensor [5], Canon [6], Riemann [7], xAct [8] and my personal choice (and the subject of this article) Cadabra. It is not my intention to provide even a cursory review of the existing computer tensor algebra systems. Instead I intend to show how Cadabra can be used to do useful work in General Relativity. I suspect that all of the results reported in this article can also be obtained using either MathTensor or xAct but as I have not done so I can not comment on their relative merits.

I chose to work with Cadabra for a number of reasons, it uses LaTeX syntax for tensor expressions, it has extensive and highly optimised routines for simplifying complex tensor expressions, it uses an interpretive language that runs on top of a C++ program, it has a small but flexible grammar leading to clear and readable code and it is in active development.

The use of (a subset of) LaTeX as the native language for Cadabra was the feature that first grabbed my attention (apart from the promise of relieving me of the tedium of complex tensor gymnastics). You may have already skimmed through this article and have wondered if I have massaged the Cadabra output for inclusion in this article. The answer is an emphatic no. *All of the output* appearing in this article was included by a 'cut-and-paste' operation directly from Cadabra's output. No stylistic changes what so ever were made to the TeX produced by Cadabra. This ability to cut and paste between Cadabra output and research documents is I believe one of Cadabra's unique strengths. This feature not only leads to readable code (we only need to be literate in a simple subset of LaTeX) but it also minimises transcription errors.

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. I will make no assumptions about the dimensionality of the space, nor the signature of the metric (other than that the metric is non-singular). However, I will assume that the connection is metric compatible (i.e. the Levi-Civita connection). For the large part I will be using abstract index notation but on the odd occasion where an explicit component based equation is given I will use a coordinate basis. You might, as a simple exercise, like to adapt some of the following Cadabra examples to work with non-coordinate basis (the changes are trivial).

2 The Cadabra software

The source code for Cadabra is freely available (but subject to the GPL licence) and has been successfully compiled on many platforms including most flavors of Linux and MacOSX. It should compile on most standard Unix like systems. The source code and binaries (for Linux) can be download from the web site [4]. Cadabra for MacOSX is best installed (by my experience) from the source via the MacPorts system [9].

Cadabra can be run either directly through the command line or through a purpose-built GUI (based on X11 and known as `xcadabra`). To run cadabra over the file `myinput.cdb` you need only type

```
cadabra < myinput.cdb > myoutput.tex
```

from the command line. The `xcadabra` GUI produces the same output but in a more convenient environment. Further details can be found in the Cadabra documentation (available through the web site) [4].

3 A simple example

How might we use Cadabra 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.

```
# --- The metric connection ------------------------------------------------

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

g_{a b}::Metric.

\partial_{#}::PartialDerivative.

cderiv:=\partial_{c}{g_{a b}} - g_{a d}\Gamma^{d}_{b c}
                              - g_{d b}\Gamma^{d}_{a c};

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

@substitute!(cderiv)(@(Gamma));

@distribute!(%);
@eliminate_metric!(%);
@canonicalise!(%);
@collect_terms!(%);
```

The output from the above code is

$$cderiv := \partial_c g_{ab} - g_{ad} \Gamma^d{}_{bc} - g_{db} \Gamma^d{}_{ac}$$

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

$$cderiv := \partial_c g_{ab} - \frac{1}{2} g_{ad} g^{de} \left(\partial_b g_{ec} + \partial_c g_{be} - \partial_e g_{bc} \right) - \frac{1}{2} g_{db} g^{de} \left(\partial_a g_{ec} + \partial_c g_{ae} - \partial_e g_{ac} \right)$$

$$\begin{aligned} cderiv := \partial_c g_{ab} &- \frac{1}{2} g_{ad} g^{de} \partial_b g_{ec} - \frac{1}{2} g_{ad} g^{de} \partial_c g_{be} + \frac{1}{2} g_{ad} g^{de} \partial_e g_{bc} \\ &- \frac{1}{2} g_{db} g^{de} \partial_a g_{ec} - \frac{1}{2} g_{db} g^{de} \partial_c g_{ae} + \frac{1}{2} g_{db} g^{de} \partial_e g_{ac} \end{aligned}$$

$$cderiv := \partial_c g_{ab} - \frac{1}{2} \partial_b g_{ac} - \frac{1}{2} \partial_c g_{ba} + \frac{1}{2} \partial_a g_{bc} - \frac{1}{2} \partial_a g_{bc} - \frac{1}{2} \partial_c g_{ab} + \frac{1}{2} \partial_b g_{ac}$$

$$cderiv := \partial_c g_{ab} - \frac{1}{2} \partial_b g_{ac} - \frac{1}{2} \partial_c g_{ab} + \frac{1}{2} \partial_a g_{bc} - \frac{1}{2} \partial_a g_{bc} - \frac{1}{2} \partial_c g_{ab} + \frac{1}{2} \partial_b g_{ac}$$

$$cderiv := 0$$

Each of these line shows selected stages of processing by Cadabra. The zero in the final line is what we were looking for – it shows that $0 = g_{ab;c}$.

We will now spend a moment of time to work through the above Cadabra code in some detail.

Comments in Cadabra are single lines that begin with the `#` character in column one. Statements in the Cadabra grammar fall into three distinct categories: *properties*, *expressions* and *algorithms*. The first three statements in the above code assign *properties* to some symbols, the next two statements define two *expressions* with names `cderiv` and `Gamma` while the remaining statements apply *algorithms* to the expressions (i.e. they perform the computations). Note that each of the statements ends with either a dot '`.`' or a semi-colon '`;`'. Only those statements that finish with a semi-colon will produce output. If you look closely at the above you will see that there are seven statements in the Cadabra code that end with a semi-colon and that there are seven statements of Cadabra output. The statements and output are in exact one-to-one correspondence. You can suppress the Cadabra output by terminating the statement with a colon '`:`' rather than the semi-colon '`;`'.

What do these statements actually mean? The first statement

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

simply declares a set of symbols that may 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.
```

The next statement

```
g_{a b}::Metric.
```

declares the symbol `g_{a b}` to represent a metric. This confers upon `g_{a b}` a raft of properties most notably that it is symmetric in its indices.

The third statement

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

assigns to the symbol `\partial` a derivative property. Note that the `#` in `\partial_{#}` signifies that any number of indices are allowed. That is both `\partial_{a}` and `\partial_{a b c d}` will be seen by Cadabra as derivative operators.

The next statement define an expression, `cderiv`.

```
cderiv:=\partial_{c}{g_{a b}} - g_{a d}\Gamma^{d}_{b c}
                              - 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. Unlike LaTeX, we must always separate the indices by one or more spaces. 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`).

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. And that argument is contained inside the pair of braces.

How do we tell Cadabra that in the expression for `cderiv` the symbols `\Gamma^{a}{}_{bc}` stands for the metric connection? That is, how do we couple the equation $\Gamma^a_{bc} = \frac{1}{2}g^{ad}\left(g_{dc,b} + g_{bd,c} - g_{bc,d}\right)$ to the expression `cderiv`? For this we create two statements, the first is an expression that defines a substitution rule, the second uses Cadabra's `@substitute` algorithm to complete the job,

```
Gamma:=\Gamma^{a}_{b c} -> (1/2) g^{a d} (  \partial_{b}{g_{d c}}
                                          + \partial_{c}{g_{b d}}
                                          - \partial_{d}{g_{b c}} );
@substitute!(cderiv)(@(Gamma));
```

The essence of this pair of statements is to substitute $\frac{1}{2}g^{ad}\left(g_{dc,b} + g_{bd,c} - g_{bc,d}\right)$ wherever the symbol `\Gamma^{a}_{b c}` appears in the expression `cderiv`. This may look simple but there are some important and subtle details that must be noted. The substitution rule `Gamma` as given above was for Γ^a_{bc} yet in the expression for `cderiv` we need Γ^d_{bc} and Γ^d_{ac}. Cadabra handles this index manipulation with ease, it will relabel dummy indices in such a way as to avoid index clashes. This feature also exists in MathTensor and xAct (but not so in GRTensorII). The construction `@(...)` is Cadabra's way of referring to the contents of an expression. The exclamation character '`!`' in the statement tells Cadabra to impose the substitution throughout the expression `cderiv`. Without the '`!`' character the substitution will be applied once at most (i.e. on the first of the two Γ^a_{bc}'s in `cderiv`).

The remaining few statements

```
@distribute!(%);
@eliminate_metric!(%);
@canonicalise!(%);
@collect_terms!(%);
```

serve only to massage the expression towards our expected result – zero. The `%` notation is Cadabra shorthand for the last computed expression, in this case `cderiv`. Each of the `@...` statements applies an algorithm to the expression `cderiv`. The algorithm `@distribute` is used to expand products, it will expand `a (b+c)` into `a b + b c`. Earlier we gave `g_{a b}` the property `::Metric`. This is used by the `@eliminate_metric` algorithm to convert combinations such as `g^{a c} g_{c b}` into the Kronecker-delta $\delta^a{}_b$. 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 on the expression. In this case it serves only to put the free indices into ascending order (look carefully at the differences between the third and second last lines of output). The final algorithm `@collect_terms` needs no explanation.

4 Covariant differentiation

Cadabra does not have predefined algorithms for computing covariant derivatives, Riemann tensors, Ricci tensor and so on. One of its strengths is that it provides a rich set of simple tools by which objects such as those just noted can be constructed. As a second example we will now see how Cadabra can be trained to compute covariant derivatives.

For a simple dual vector such as v_a the textbook definition of the covariant derivative $v_{a;b}$ is

$$v_{a;b} = v_{a,b} - \Gamma^c_{ab} v_c$$

Here we will pursue a slight variation, one that lends itself well to computing higher order covariant derivatives in Cadabra.

Choose any point P in the spacetime and construct any curve through that point. Let D^a be the unit tangent vector to the curve, A^a be a parallel vector field along the curve and let the curve be parametrised by the proper distance s. Thus we have

$$\frac{dv_a}{ds} = v_{a,b} D^b$$

$$0 = \nabla_D A^a = \frac{dA^a}{ds} + \Gamma^a_{bc} A^b D^c$$

Since $v_a A^a$ is a scalar function of s we can compute its derivative along the curve in two ways, by ordinary differentiation in s or by applying the Leibniz rule to $\nabla_D (v_a A^a)$. This leads to

$$v_{a;b} A^a D^b = \frac{d(v_a A^a)}{ds}$$

The left hand side is what we are looking for, while the right hand side is something we can easily train Cadabra to perform. The final code contains two new algorithms, `@prodsort` and `@factor_out`. Their actions are very simple, `\@prodsort` rewrites any expression so that the symbols appear in a particular order (usually in alphabetical order though this can be overridden with the `::SortOrder` property) and `@factor_out` factors out nominated symbols from an expression. So, without further ado here is the code.

```
# --- Covariant differentiation -------------------------------------------------

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

\partial_{#}::PartialDerivative.

# --- construct the scalar v_{a} A^{a}  -----------------------------------------

scalar:=v_{a} A^{a}:

# --- compute the derivative  ---------------------------------------------------

derivD:=D^{c}\partial_{c}{@(scalar)}:
@distribute!(%):
@prodrule!(%):
@distribute!(%):
@substitute!(%)(D^{a}\partial_{a}{A^{b}} -> -\Gamma^{b}_{ac}A^{a}D^{c}):
@substitute!(%)(D^{a}\partial_{a}{D^{b}} -> -\Gamma^{b}_{ac}D^{a}D^{c}):
@prodsort!(%):
@rename_dummies!(%):
@canonicalise!(%):

# --- tidy up and display the results  ------------------------------------------

@factor_out!(scalar){A^{a}}:
@factor_out!(scalar){D^{a}};

@factor_out!(derivD){A^{a}}:
@factor_out!(derivD){D^{a}};
```

The above code produces the following output

$$scalar \ := v_a A^a$$

$$derivD \ := A^a D^b \left(\partial_b v_a - \Gamma^b{}_{ac} v_b\right)$$

The right hand side is what we were seeking, $A^a D^b v_{a;b}$. As this quantity is also a scalar it could be used as a starting point for further rounds of differentiation. Clearly this is easy to do and would lead to expressions for higher order covariant derivatives of v_a. This is the basis for the following example.

5 Covariant differentiation and the Riemann tensor

The somewhat unorthodox way in which we computed the covariant derivative in the previous example is actually very well suited to our next example – computing the Riemann tensor by commutation of successive covariant derivatives. In this example we will 'discover' that

$$R_a{}^d{}_{bc} = \partial_b \Gamma^d{}_{ac} - \partial_c \Gamma^d{}_{ab} + \Gamma^d{}_{be}\Gamma^e{}_{ac} - \Gamma^d{}_{ce}\Gamma^e{}_{ab}$$

follows directly from $v_{a;b;c} - v_{a;c;b} = -R_a{}^d{}_{bc} v_d$

Here we will be using two covariant derivatives and thus, in the spirit of the previous example, we imagine having two distinct curves through the point P. Let D^a and E^a be the respective unit tangent vectors and let u and v be the respective proper distances along the curves (I could have retained s for the first curve but for aesthetics I prefer to use u and v here). This example is slightly more complicated than the previous example but we still have considerable freedoms to choose the nature of the vector fields A^a, D^a and E^a along the two curves through P. We will do the calculations in two parts, first we will compute $d(d(v_a A^a)/du)dv$ then we reverse the order $d(d(v_a A^a)/dv)du$. So for the first part we choose to parallel transport A^a and D^a along E^a. In the second part we simply swap D^a and E^a. Thus we have

$$\frac{dv_a}{du} = v_{a,b} D^b \,, \qquad \frac{dv_a}{dv} = v_{a,b} E^b$$

$$0 = \nabla_D \, A^a = \frac{dA^a}{du} + \Gamma^a_{bc} A^b D^c$$

$$0 = \nabla_E \, A^a = \frac{dA^a}{dv} + \Gamma^a_{bc} A^b E^c$$

$$0 = \nabla_E \, D^a = \frac{dD^a}{dv} + \Gamma^a_{bc} D^b E^c$$

$$0 = \nabla_D \, E^a = \frac{dE^a}{dv} + \Gamma^a_{bc} E^b D^c$$

and this leads to

$$\left(v_{a;b;c} - v_{a;c;b}\right) A^a D^b E^c = \left(\frac{d}{du}\frac{d}{dv} - \frac{d}{dv}\frac{d}{du}\right)\left(v_a A^a\right)$$

The right hand side is very easy to implement in Cadabra – just two (almost identical) rounds of the code given in the previous example. Here is the Cadabra code.

```
# --- covariant differentiation and the Riemann tensor -----------------------

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

\partial_{#}::PartialDerivative.

# --- force Gamma to be symmetric in its lower two indices -------------------

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

# --- construct the scalar v_{a} A^{a}  --------------------------------------

scalar:=v_{a} A^{a}:

# --- compute the covariant derivative in the direction of D^a  --------------

derivD:=D^{c}\partial_{c}{@(scalar)}:
@distribute!(%):
@prodrule!(%):
@distribute!(%):
@substitute!(%)(D^{a}\partial_{a}{A^{b}} -> -\Gamma^{b}_{a c}A^{a}D^{c}):
@prodsort!(%):
@rename_dummies!(%):
@canonicalise!(%):

# --- compute the covariant derivative in the direction of E^a  --------------

derivDE:=E^{c}\partial_{c}{@(derivD)}:
@distribute!(%):
@prodrule!(%):
@distribute!(%):
@substitute!(%)(E^{a}\partial_{a}{A^{b}} -> -\Gamma^{b}_{a c}A^{a}E^{c}):
@substitute!(%)(E^{a}\partial_{a}{D^{b}} -> -\Gamma^{b}_{a c}D^{a}E^{c}):
@prodsort!(%):
@rename_dummies!(%):
@canonicalise!(%);

# --- copy to derivED then swap the order of the derivatives -----------------

derivED:=@(derivDE):

@substitute!(%)(E^{a} -> F^{a}):
@substitute!(%)(D^{a} -> E^{a}):
@substitute!(%)(F^{a} -> D^{a});

# --- compute difference in mixed covariant derivatives ----------------------

diff:=@(derivDE) - @(derivED):

# --- tidy up and display the results  ---------------------------------------

{A^{a},D^{a},E^{a},v_{a},\Gamma^{a}_{b c}}::SortOrder.

@prodsort!(%):
@rename_dummies!(%):
@canonicalise!(%):
@collect_terms!(%):

@factor_out!(%){A^{a}}:
@factor_out!(%){D^{a}}:
```

```
@factor_out!(%){E^{a}}:
@factor_out!(%){v_{a}};

@print["A^a D^b E^c (v_{a;b;c}-v_{a;c;b})="~@(diff)];
```

Here is the output form the above code.

$$\begin{aligned} derivDE \;:=\; & -A^a D^b E^c \Gamma^d{}_{ac} \partial_b v_d - A^a D^b E^c \Gamma^d{}_{bc} \partial_d v_a + A^a D^b E^c \partial_{bc} v_a + A^a D^b E^c \Gamma^d{}_{ac} \Gamma^e{}_{bd} v_e \\ & + A^a D^b E^c \Gamma^d{}_{ae} \Gamma^e{}_{bc} v_d - A^a D^b E^c \partial_c \Gamma^d{}_{ab} v_d - A^a D^b E^c \Gamma^d{}_{ab} \partial_c v_d \end{aligned}$$

$$\begin{aligned} derivED \;:=\; & -A^a E^b D^c \Gamma^d{}_{ac} \partial_b v_d - A^a E^b D^c \Gamma^d{}_{bc} \partial_d v_a + A^a E^b D^c \partial_{bc} v_a + A^a E^b D^c \Gamma^d{}_{ac} \Gamma^e{}_{bd} v_e \\ & + A^a E^b D^c \Gamma^d{}_{ae} \Gamma^e{}_{bc} v_d - A^a E^b D^c \partial_c \Gamma^d{}_{ab} v_d - A^a E^b D^c \Gamma^d{}_{ab} \partial_c v_d \end{aligned}$$

$$diff \;:=\; A^a D^b E^c v_d \left(\Gamma^d{}_{be} \Gamma^e{}_{ac} - \partial_c \Gamma^d{}_{ab} - \Gamma^d{}_{ce} \Gamma^e{}_{ab} + \partial_b \Gamma^d{}_{ac} \right)$$

$$A^a D^b E^c \left(v_{a;b;c} - v_{a;c;b} \right) = A^a D^b E^c v_d \left(\Gamma^d{}_{be} \Gamma^e{}_{ac} - \partial_c \Gamma^d{}_{ab} - \Gamma^d{}_{ce} \Gamma^e{}_{ab} + \partial_b \Gamma^d{}_{ac} \right)$$

The last line of output is what we have been seeking and apart from it being mathematically correct it is worth noting how that line was created. The right hand side was obtained from a result computed by Cadabra while the left hand side was a string supplied by us. This construction is performed in the `@print[...]` algorithm. You can see it in the last line of the Cadabra code. Its use is clear – it creates nicely formatted output. But there is a potential danger here – the onus falls on the user (us) to ensure that the left hand side is what it should be, Cadabra does no processing what so ever on the argument to `@print`. So if you chose to write nonsense such as

```
@print["A_a + B_c = g_{ab}"];
```

then Cadabra will dutifully obey.

The output for this example could have been restricted to just the final line (by changing the appropriate '`;`' to '`:`'). But there is value in including output from other lines for it shows how the computations unfold and it allows us to verify that Cadabra is doing what we think and want it to do. Output like this is very useful when first writing and later debugging Cadabra code.

There is one very important part of the calculations that we have not yet spoken about – How do we tell Cadabra that we are using a torsion free connection? The line in the above code that does the job is

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

This tells Cadabra that we want the connection to be symmetric in its lower two indices (i.e. torsion free).

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.

If Young diagrams and tableaux are not your cup of tea 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 Γ^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 (assuming `%` is an expression that contains Γ^a_{bc})

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

G_{a b}::Symmetric.

@substitute!(%)(\Gamma^{a}_{b c} ->  G^{a} G_{b c});
@prodsort!(%):
@canonicalise!(%):
@collect_terms!(%):
@substitute!!(%)(G^{a} G_{b c} -> \Gamma^{a}_{b c});
```

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 Γ^a_{bc}. If on the other hand you can be sure that such problems can not arise (e.g. you apply the trick after all the derivatives have been computed) 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 $A_{abcde} = G_{ab}G_{cde}$).

6 The Gauss equation

Lest it be thought that every covariant derivative in Cadabra needs to be cast in the form given in the previous examples, here is an application of Cadabra to the derivation of the Gauss equation relating the induced and ambient curvatures of a hypersurface in an $n-$dimensional Riemannian manifold.

We shall start with a brief review of the underlying mathematics. 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 we will denote 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. $v_a n^a = 0$), we have

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

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 [10]).

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

We will start with 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 ∇. We will use the symbol `hab` to record the projection operator and `vpq` to record the covariant derivative $D_q v_p$. Thus our 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}}:
```

We will also need an expression for the commutation of the covariant derivatives, $(D_r D_q - D_q D_r)\, v_p$ which we write 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)):
```

Finally we will need to introduce some standard substitutions to simplify and tidy the result. Note that *all of the previous definitions and following substitutions are exactly what we would normally do if we were to do these calculations by hand.* For example, the lines

```
@substitute!(%)(h^{a}_{b} n^{b} -> 0):
@substitute!(%)(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!(%)(\nabla_{a}{g^{b}_{c}} -> 0):
```

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

```
@substitute!(%)(n^{a}\nabla_{b}{v_{a}} -> -v_{a}\nabla_{b}{n^{a}}):
```

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

```
@substitute!(%)(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!(%)(h^{p}_{a}h^{q}_{b}\nabla_{p}{n_{q}} -> K_{a b}):
@substitute!(%)(h^{p}_{a}h^{q}_{b}\nabla_{p}{n^{b}} -> K_{a}^{q}):
```

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

Clearly we have to supplement the above code fragments 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.

```
# --- The Gauss equation ----------------------------------------------------

::PostDefaultRules( @@collect_terms!(%) ).
{a,b,c,d,e,f,g,i,j,k,l,m,n,o,p,q,r,s,t,u#}::Indices(position=fixed).

\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!(%):
@prodrule!(%):
@distribute!(%):
@eliminate_kr!(%):

# --- standard substitutions ------------------------------------------------

@substitute!(%)(h^{a}_{b} n^{b} -> 0):
@substitute!(%)(h^{a}_{b} n_{a} -> 0):
@substitute!(%)(\nabla_{a}{g^{b}_{c}} -> 0):
@substitute!(%)(n^{a}\nabla_{b}{v_{a}} -> -v_{a}\nabla_{b}{n^{a}}):
@substitute!(%)(v_{a}\nabla_{b}{n^{a}} -> v_{p}h^{p}_{a}\nabla_{b}{n^{a}}):
@substitute!(%)(h^{p}_{a}h^{q}_{b}\nabla_{p}{n_{q}} -> K_{a b}):
@substitute!(%)(h^{p}_{a}h^{q}_{b}\nabla_{p}{n^{b}} -> K_{a}^{q}):

# --- tidy up and display the results ---------------------------------------

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

@prodsort!(%):
@rename_dummies!(%):
@canonicalise!(%):
@factor_out!!(%){h^{a}_{b}};
```

The output returned by Cadabra is

$$vpqr := h^{a}{}_{p} h^{b}{}_{q} h^{c}{}_{r} \left(\nabla_c \nabla_b v_a - \nabla_b \nabla_c v_a\right) + K_{pr} K_q{}^{a} v_a - K_{pq} K_r{}^{a} v_a$$

which, although correct, is not in the form we are most familiar with. We could reformat the output by including suitable `@print` statements or we could wrest control from Cadabra and do the final tidying up by hand (we are not invalids and it's wise to retain our skills).

Recall that `vpqr` was defined to be $(D_r D_q - D_q D_r)\, v_p$. If we use this and the Ricci identity for curvatures $v_{a;b;c} - v_{a;c;b} = -R_a{}^d{}_{bc} v_d$ we very easily recover the Gauss equation in (one of) its familiar forms

$$\left({}^h R_p{}^s{}_{qr}\right) = h^a{}_p h^s{}_d h^b{}_q h^c{}_r \left({}^g R_a{}^d{}_{bc}\right) + K_{pr} K_q{}^s - K_{pq} K_r{}^s$$

7 Outlook

Simple problems such as those given here can be done easily by hand (or passed off to a graduate student). But they tell us nothing about Cadabra's utility or performance for large scale tensor computations in General Relativity. In a follow-up paper [11] we will report on the use of Cadabra for higher order Riemann normal expansions of the metric, connection and solutions to the initial and boundary value problems for the geodesic equations. The results have been completed up to 6th order in the curvature tensor.

It is my opinion that Cadabra is very well suited to large scale tensor algebra computations in General Relativity and will be a welcome addition to the relativist's computational toolkit.

8 Acknowledgements

I am very grateful to Kasper Peeters for his many helpful suggestions. Any errors, omissions or inaccuracies in regard to Cadabra are entirely my fault (I hope there are none).

References

[1] P. Musgarve, D. Pollney, and K. Lake. GRTensorII home page. http://grtensor.phy.queensu.ca/.

[2] K. Peeters, A field-theory motivated approach to symbolic computer algebra, arXiv:cs/0608005v2.

[3] K. Peeters, Introducing Cadabra: a symbolic computer algebra system for field theory problems, arXiv:hep-th/0701238.

[4] K. Peeters. The Cadabra home page. http://www.aei.mpg.de/~peekas/cadabra/.

[5] S. M. Christensen and L. Parker. MathTensor home page. http://smc.vnet.net/MathTensor.html.

[6] L. R. U. Manssur and R. Portugal, The canon package: a fast kernel for tensor manipulators, *Computer Physics Communications* **157** (2004) no. 2, 173–180. http://www.lncc.br/~portugal/Invar.html.

[7] R. Portugal and S. L. Sautú, Applications of Maple to general relativity, *Computer Physics Communications* **105** (1997) no. 2-3, 233–253.

[8] J. Martín-García. xAct: Efficient tensor computer algebra for Mathematica. http://metric.iem.csic.es/Martin-Garcia/xAct/.

[9] MacPorts. The MacPorts Project. http://www.macports.org/.

[10] I. Chavel, *Riemannian Geometry. A modern introduction, 2nd ed.* Cambridge University Press, Cambridge., 2006.

[11] L. Brewin, Riemann Normal Coordinate expansions using Cadabra. In preparation, 2009.