October 25, 2024

The Cadabra Book

A field-theory motivated approach to symbolic computer algebra

Kasper Peeters

kasper.peeters@cadabra.science

0 Contents

1 Introduction and overview

1.1 Bird's eye overview

Cadabra is a symbolic computer algebra system (CAS) designed to solve problems in physics, in particular (but not limited to) those which deal with classical and quantum field theory. Its input format is a subset of TeX, which means that throughout your computations the maths will all stay in a form which is hopefully already familiar to you. It has a large number of facilities which make it easy to work with tensors, anti-commuting objects, implicit indices, implicit coordinate dependence and so on; things which help to keep mathematical expressions compact and readable, so that your notebooks resemble what you would do with pen-and-paper.

Cadabra is at its core a Python module (written in C++ and exposed to Python using pybind11), which also contains a pre-parser which turns Cadabra's language into pure Python. You can use through a command line client, a graphical notebook interface, or via Jupyter through the Cadabra kernel. You can, if you want to, also use it from Python directly, or as a C++ library.

1.2 Cadabra's design philosophy

Cadabra is built around the fact that many computations do not have one single and unique path between the starting point and the end result. When we do computations on paper, we often take bits of an expression apart, do some manipulations on them, stick them back into the main expression, and so on. Often, the manipulations that we do are far from uniquely determined by the problem, and often there is no way even in principle for a computer to figure out what is 'the best' thing to do.

What we need the computer to do, in such a case, is to be good at performing simple but tedious steps, without enforcing on the user how to do a particular computation. In other

words, we want the computer algebra system to be a scratchpad, leaving us in control of which steps to take, not forcing us to return to a 'canonical' expression at every stage.

Most existing computer algebra systems allow for this kind of work flow only by requiring to stick clumsy 'inert' or 'hold' arguments onto expressions, by default always 'simplifying' every input to some form they think is best. Cadabra starts from the other end of the spectrum, and as a general rule keeps your expression untouched, unless you explicitly ask for something to be done to it.

Another key issue in the design of symbolic computer algebra systems has always been whether or not there should be a distinction between the 'data language' (the language used to write down mathematical expressions), the 'manipulation language' (the language used to write down what you want to do with those expressions) and the 'implementation language' (the language used to implement algorithms which act on mathematical expressions). Many computer algebra systems take the approach in which these languages are the same (Axiom, Reduce, Sympy) or mostly the same apart from a small core which uses a different implementation language (Mathematica, Maple). The Cadabra project is rooted in the idea that for many applications, it is better to keep a clean distinction between these three languages. Cadabra writes mathematics using LaTeX, is programmable in Python, and is under the hood largely written in C++.

1.3 History

Cadabra was originally written around 2001 to solve a number of problems related to higher-derivative supergravity [1, 2]. It was then expanded and polished, and first saw its public release in 2007 [3]. During the years that followed, it became clear that several design decisions were not ideal, such as the use of a custom programming language and the lack of functionality for component computations. Over the course of 2015-2016 a large rewrite took place, which resulted in Cadabra 2.x [4]. This new version is programmable in Python and does both abstract and component computations. From 2017 to 2022 Dominic Price joined the team. He was responsible for many improvements, such as the add-on package system, the conversion to pybind11 for the Python bindings, the implementation of the `meld` algorithm and many others. From 2022 onwards the emphasis has been on making it easier to run Cadabra (for instance by the introduction of a publically available Jupyter server with the Cadabra kernel, and by making it much easier to install the software locally on all platforms). From 2024 the focus is again on adding more physics functionality as well as tutorials.

2 The input format

2.1 Input format

2.1.1 Mathematical expressions

The input format of Cadabra is closely related to the notation used by LaTeX to denote tensorial expressions. That is, one can use not only bracketed notation to denote child objects, like in

```
object[child,child]
```

but also the usual sub- and superscript notation like

```
object^{child child}_{child}
```

One can use backslashes in the names of objects as well, just as in LaTeX. All of the symbols that one enters this way are considered "passive", that is, they will go into the expression tree just like one has entered them.

Expressions are entered by using the ':=' operator, as in

```
ex:=A+B+C_{m} C^{m};
```

$$A + B + C_m C^m$$

Expressions (the 'ex' above) are ordinary Python objects (of type `cadabra2.Ex`), and their names can thus only contain normal alphanumeric symbols.

```
type(ex);
```

```
<class 'cadabra2.Ex'>
```

Lines always have to be terminated with either a ";" or a ":". These delimiting symbols act in the same way as in Maple: the second form suppresses the output of the entered expression. Long expressions can, because of these delimiters, be spread over many subsequent input lines. Any line starting with a "#" sign is considered to be a comment (even when it appears within a multi-line expression). Comments are always ignored completely (they do not end up in the expression tree. When entering maths as above (using the ':=' assignment operator) you do not need to indicate that the right-hand side is mathematics. There are situations, however, when you do need to give Cadabra a hint that what you type is maths, not Python. In this case, you add dollar symbols, just as in LaTeX,

```
substitute($A + B + C$, $C -> D$);
```

$$A + B + D$$

As you can see, this uses an 'inline' definition of a mathematical expression, without giving it a name.

2.1.2 Algorithms

Algorithms are ordinary Python functions, which act on `cadabra2.Ex` objects.

2.2 Printing expressions in various formats

2.2.1 Basic usage

With basic use of Cadabra, you will typically display your expressions by postfixing them with a semi-colon, as in

```
ex:=A_{m n} ( B^{n} + 3 C^{n} );
```

$$A_{mn}\left(B^{n} + 3C^{n}\right)$$

What happens behind the scenes is that the semi-colon gets translated to a call of `display` on the last-entered expression. It is therefore equivalent to

```
display(ex)
```

$$A_{mn}\left(B^{n} + 3C^{n}\right)$$

If you do not want to display the expression, post-fix with a colon, as in

```
ex:=A_{m n} ( B^{n} + 3 C^{n} ):
```

If you want to display an expression again later, you can just write the name of the expression followed by a semi-colon, or use the `display` function again,

```
ex;
display(ex)
```

$$A_{mn}\left(B^{n} + 3C^{n}\right)$$

$$A_{mn}\,(B^n + 3C^n)$$

Note that while it may be tempting to use `print(ex)`, the `display` function is better because it knows about the capabilities of the interface used, and it will automatically select a text output when you use Cadabra from the terminal, or LaTeX output when you use it in the graphical notebook.

2.2.2 Other output formats

Cadabra expressions are standard Python objects, and as such they have a `__str__` method which converts them into a printable expression, and a `__repr__` method to produce a machine readable form. These are called by the standard `str` and `repr` Python functions, as the examples below show.

```
print(str(ex))
```

```
\begin{verbatim}A_{m n} (B^{n} + 3C^{n})
\end{verbatim}
```

```
print(repr(ex))
```

```
\begin{verbatim}\prod(A_{m n})(\sum(B^{n})(3C^{n}))
\end{verbatim}
```

In addition there are some methods to obtain output useful in other software: Mathematica, LaTeX and Sympy:

```
print(ex.mma_form())
```

```
\begin{verbatim}A[DNm, DNn]*(B[UPn]+3*C[UPn])
\end{verbatim}
```

```
print(ex._latex_())
```

```
\begin{verbatim}A_{m n} \brwrap{(}{B^{n}+3C^{n}}{)}
\end{verbatim}
```

```
print(ex.sympy_form())
```

```
\begin{verbatim}A(DNm, DNn)*(B(UPn)+3*C(UPn))
\end{verbatim}
```

2.2.3 Printing custom LaTeX

If you want to make sure that a string which you have created ”by hand” will be processed by LaTeX and display in typeset form, use the ‘`LaTeXString`‘ object. This is essentially a normal string, but ”tagged” with the information that it contains LaTeX code.

```
s = LaTeXString(r"\int_{-\infty}^\infty␣\exp\brwrap{[}{␣-a␣x^2␣}{]}␣{\rm␣d}␣x")
    ;
```

$$\int_{-\infty}^{\infty} \exp\left[-ax^2\right] \mathrm{d}x$$

2.3 Object properties and declaration

2.3.1 Generic properties

Symbols in Cadabra have no a-priori "meaning". If you write `\Gamma`, the program will not know that it is supposed to be, for instance, a Clifford algebra generator. You will have to declare the properties of symbols, i.e. you have to tell Cadabra explicitly that if you write `\Gamma`, you actually mean a Clifford algebra generator. This indirect way of attaching a meaning to a symbol has the advantage that you can use whatever notation you like; if you prefer to write `\gamma`, or perhaps even `\rho` if your paper uses that, then this is perfectly possible (object properties are a bit like "attributes" in `Mathematica` or "domains" in `Axiom` and `MuPAD`).

Properties are all written using capitals to separate words, as in `AntiSymmetric`. This makes it easier to distinguish them from commands. Properties of objects are declared by using the "::" characters. This can be done "at first use", i.e. by just adding the property to the object when it first appears in the input. As an example, one can write

```
F_{m n p}::AntiSymmetric;
```

Attached property AntiSymmetric to F_{mnp}.

This declares the object to be anti-symmetric in its indices. The property information is stored separately, so that further appearances of the "`F_{m n p}`" object will automatically share this property. A list of all properties is available from the manual pages on the web site.

Note that properties are attached to patterns, Therefore, you can have

```
R_{m n}::Symmetric;
R_{m n p q}::RiemannTensor;
```

Attached property Symmetric to R_{mn}.

Attached property TableauSymmetry to R_{mnpq}.

at the same time. The program will not warn you if you use incompatible properties, so if you make a declaration like above and then later on do

```
R_{m n}::AntiSymmetric;
```

Attached property AntiSymmetric to R_{mn}.

this may lead to undefined results. The fact that objects are attached to patterns also means that you can use something like wildcards. In the following declaration,

```
{m#, n#}::Indices(vector);
```

Attached property Indices(position=free) to $[m\#,\ n\#]$.

the entire infinite set of objects $m1, m2, m3, \ldots$ and $n1, n2, n3, \ldots$ are declared to be in the dummy index set "vector" (this way of declaring ranges of objects is similar to the "autodeclare" declaration method of FORM). Properties can be assigned to an entire list of symbols with one command, namely by attaching the property to the list. For example,

```
{n, m, p, q}::Integer(1..d);
```

Attached property Integer to $[n,\ m,\ p,\ q]$.

This associates the property "Integer" to each and every symbol in the list. However, there is also a concept of "list properties", which are properties which are associated to the list as a whole. Examples of list properties are "AntiCommuting" or "Indices". Objects can have more than one property attached to them, and one should therefore not confuse properties with the "type" of the object. Consider for instance

```
x::Coordinate;
W_{m n p q}::WeylTensor;
W_{m n p q}::Depends(x);
```

Attached property Coordinate to x.

Attached property TableauSymmetryWeylTensor to W_{mnpq}.

Attached property Depends to W_{mnpq}.

This attaches two completely independent properties to the pattern W_{mnpq}. In the examples above, several properties had arguments (e.g. "`vector`" or "`1..d`"). The general form of these arguments is a set of key-value pairs, as in

```
T_{m n p q}::TableauSymmetry(shape={2,1}, indices={0,2,1});
```

Attached property TableauSymmetry to T_{mnpq}.

In the simple cases discussed so far, the key and the equal sign was suppressed. This is allowed because one of the keys plays the role of the default key. Therefore, the following two are equivalent,

```
{ m, n }::Integer(range=0..d);
{ m, n }::Integer(0..d);
```

Attached property Integer to $[m,\ n]$.

Attached property Integer to $[m,\ n]$.

See the detailed documentation of the individual properties for allowed keys and the one which is taken as the default. Finally, there is a concept of "inherited properties". Consider e.g. a sum of spinors, declared as

```
{\psi_1, \psi_2, \psi_3}::Spinor;
ex:= \psi_1 + \psi_2 + \psi_3;
```

Attached property Spinor to $[\psi_1,\ \psi_2,\ \psi_3]$.

$\psi_1 + \psi_2 + \psi_3$

Here the sum has inherited the property "Spinor", even though it does not have normal or intrinsic property of this type. Properties can also inherit from each other, e.g.

```
\Gamma_{#}::GammaMatrix.
ex:=\Gamma_{p o i u y};
```

Γ_{poiuy}

```
canonicalise(_);
```

$-\Gamma_{iopuy}$

The `GammaMatrix` property inherits from `AntiSymmetric` property, and therefore the `\Gamma` object is automatically anti-symmetric in its indices. indices.

2.3.2 List properties and symbol groups

Some properties are not naturally associated to a single symbol or object, but have to do with collections of them. A simple example of such a property is `AntiCommuting`. Although it sometimes makes sense to say that "ψ_m is anticommuting" (meaning that $\psi_m\psi_n = -\psi_n\psi_m$), it happens just as often that you want to say "ψ and χ anticommute" (meaning that $\psi\chi = -\chi\psi$). The latter property is clearly relating two different objects.

Another example is dummy indices. While it may make sense to say that "m is a dummy index", this does not allow the program to substitute m with another index when a clash of dummy index names occurs (e.g. upon substitution of one expression into another). More useful is to say that "m, n, and p are dummy indices of the same type", so that the program can relabel a pair of m's into a pair of p's when necessary.

In Cadabra such properties are called "list properties". You can associate a list property to a list of symbols by simply writing, e.g. for the first example above,

```
{ \psi, \chi }::AntiCommuting;
```

Attached property AntiCommuting to $[\psi,\ \chi]$.

Note that you can also attach normal properties to multiple symbols in one go using this notation. The program will figure out automatically whether you want to associate a normal property or a list property to the symbols in the list.

Lists are ordered, although the ordering does not necessarily mean anything for all list properties (it is relevant for e.g. `SortOrder` but irrelevant for e.g. `AntiCommuting`).

2.3.3 Querying properties

For many built-in algorithms, assigning properties to the objects which appear in your expressions will be enough to make them work. However, sometimes you may want to ex-

plicitly query whether a particular symbol has a particular property. The following example shows how this works.

```
A_{m n}::Symmetric;
if Symmetric.get($A_{m n}$):
   print("A_{m␣n}␣is␣symmetric.")
if AntiSymmetric.get($A_{m n}$):
   print("A_{m␣n}␣is␣anti-symmetric.")
```

Property Symmetric attached to A_{mn}.

```
\begin{verbatim}A_{m n} is symmetric.
\end{verbatim}
```

The object returned by `Property.get(...)` is either `None` or the property which you asked about. It is possible to do something with that property, e.g. attach it to another symbol. In the example below, we start off with one tensor with a symmetry, and then attach it to another symbol.

```
A_{m n p}::TableauSymmetry(shape={1,1}, indices={1,2});
p = TableauSymmetry.get($A_{m n p}$)
p.attach($B_{m n p}$)
ex:= B_{m n p} - B_{m p n};
canonicalise(_);
```

Property TableauSymmetry attached to A_{mnp}.

$B_{mnp} - B_{mpn}$

$2B_{mnp}$

2.4 Indices, dummy indices and automatic index renaming

In Cadabra, all objects which occur as subscripts or superscripts are considered to be "indices". The names of indices are understood to be irrelevant when they occur in a pair, and automatic relabelling will take place whenever necessary in order to avoid index clashes.

Cadabra knows about the differences between free and dummy indices. It checks the input for consistency and displays a warning when the index structure does not make sense. Thus, the input

```
ex:= A_{m n} + B_{m} = 0;
```

will results in an error message. The location of indices is, by default, not considered to be relevant. That is, you can write

```
{m, n}::Indices(name="free");
ex:=A_{m} + A^{m};
```

Attached property Indices(position=free) to $[m,\ n]$.

$A_m + A^m$

as input and these are considered to be consistent expressions. You can collect such terms by using `lower_free_indices` or `raise_free_indices`,

```
lower_free_indices(ex);
```

$2A_m$

If, however, the position of an index means something (like in general relativity, where index lowering and raising implies contraction with a metric), then you can declare index positions to be "fixed". This is done using

```
{a,b,c,d,e,f}::Indices(name="fixed", position=fixed);
```

Attached property Indices(position=fixed) to $[a,\ b,\ c,\ d,\ e,\ f]$.

Cadabra will raise or lower indices on such expressions to a canonical form when the `canonicalise` algorithm is used,

```
ex:= G_{a b} F^{a b} + G^{a b} F_{a b};
canonicalise(_);
```

$G_{ab}F^{ab} + G^{ab}F_{ab}$

$2G^{ab}F_{ab}$

If upper and lower indices should remain untouched at all times, there is a third index position type, called 'independent',

```
{q,r,s}::Indices(name="independent", position=independent);
ex:= G_{q r} F^{q r} + G^{q r} F_{q r};
canonicalise(_);
```

Attached property Indices(position=independent) to $[q,\ r,\ s]$.

$G_{qr}F^{qr} + G^{qr}F_{qr}$

$G_{qr}F^{qr} + G^{qr}F_{qr}$

As the last line shows, the index positions have remained unchanged. When substituting an expression into another one, dummy indices will automatically be relabelled when necessary. To see this in action, consider the following example:

```
ex:= G_{a b} Q;
rl:= Q-> F_{a b} F^{a b};
substitute(ex, rl);
```

$G_{ab}Q$

$Q \to F_{ab}F^{ab}$

$G_{ab}F_{cd}F^{cd}$

The a and b indices have automatically been relabelled to c and d in order to avoid a conflict with the free indices on the G_{ab} object. You may have noticed that when you write `T_{a b}` the '`a b`' in the subscript is not interpreted as a product, but rather as two different indices to the tensor T.

2.5 Implicit versus explicit indices

When writing expressions which involves vectors, spinors and matrices, one often employs an implicit notation in which some or all of the indices are suppressed. Examples are

$$a = Mb\,, \quad \overline{\psi}\gamma^m\chi\,,$$

where a and b are vectors, ψ and χ are spinors and M and γ^m are matrices. Clearly, the computer cannot know this without some further information. In Cadabra objects can carry implicit indices, through the `ImplicitIndex` property. There are derived forms of this, e.g. `Matrix` and `Spinor`. The following example shows how implicit indices ensure that objects do not get moved through each other when sorting expressions.

```
{a,b}::ImplicitIndex;
M::Matrix;
ex:= a = M b;
sort_product(_);
```

Attached property ImplicitIndex to $[a,\ b]$.

Attached property Matrix to M.

$a = Mb$

$a = Mb$

If you had not made the property assignment in the first two lines, the `sort_product` would have incorrectly swapped the matrix and vector, leading to a meaningless expression.

If you have more than one set of implicit indices, it is best to use a form of `ImplicitIndex` which makes explicit which indices are suppressed. In the following example, we write consider the expression Mcb in which M is a matrix acting on the vector b, while c is a different matrix which does not act on the same vector space. In other words, we consider ${M^i}_j{c^m}_n b^j$. Clearly we can also write this as Mbc, which is indeed what `sort_product` converts it to.

```
{i,j}::Indices(vector);
{m,n}::Indices(spinor);
M::ImplicitIndex(M^{i}_{j});
```

```
b::ImplicitIndex(b^{i});
c::ImplicitIndex(c^{m}_{n});
```

Attached property Indices(position=free) to $[i,\ j]$.

Attached property Indices(position=free) to $[m,\ n]$.

Attached property ImplicitIndex to M.

Attached property ImplicitIndex to b.

Attached property ImplicitIndex to c.

```
ex:= M c b;
```

Mcb

```
sort_product(_);
```

Mbc

Such explicit property information is also respected by operators like `Trace`. The following example shows how to remove objects from traces when they do not carry any indices on which the trace acts.

```
Tr{#}::Trace(indices=vector);
ex:= Tr( M c M );
untrace(_);
```

Attached property Trace to $Tr\,(\#)$.

$Tr\,(McM)$

$cTr\,(MM)$

2.5.1 Converting between implicit and explicit

It is possible to convert from implicit indices to explicit indices, that is, make Cadabra write out all implicit indices explicitly. For this to work you need to have declared an `ImplicitIndex` property which lists the explicit indices of the object. Cadabra will then take care of creating index lines.

```
{i,j,k}::Indices;
a::ImplicitIndex(a^{i});
M::ImplicitIndex(M^{i}_{j});
ex:= M a;
```

Attached property Indices(position=free) to $[i,\ j,\ k]$.

Attached property ImplicitIndex to a.

Attached property ImplicitIndex to M.

$$Ma$$

```
explicit_indices(ex);
```

$$M^{i}{}_{j}a^{j}$$

Note how dummy indices were introduced automatically.

3 Mathematical properties

3.1 Derivatives and implicit dependence on coordinates

There is no fixed notation for derivatives; as with all other objects you have to declare derivatives by associating a property to them, in this case the `Derivative` property.

```
\nabla{#}::Derivative;
```

Property Derivative attached to $\nabla\{\#\}$.

Derivative objects can be used in various ways. You can just write the derivative symbol, as in

```
ex:=\nabla{ A_{\mu} };
```

∇A_{μ}

Or you can write the coordinate with respect to which the derivative is taken,

```
s::Coordinate;
A_{\mu}::Depends(s);
ex:=\nabla_{s}{ A_{\mu} };
```

Property Coordinate attached to s.

Property Depends attached to A_{μ}.

$\nabla_{s} A_{\mu}$

Finally, you can use an index as the subscript argument, as in

```
{ \mu, \nu }::Indices(vector);
ex:=\nabla_{\nu}{ A_{\mu} };
```

Property Indices(position=free) attached to $[\mu, \nu]$.

$\nabla_\nu A_\mu$

(in which case the first line is, for the purpose of using the derivative operator, actually unnecessary). The main point of associating the `Derivative` property to an object is to make the object obey the Leibnitz or product rule, as illustrated by the following example,

```
\nabla{#}::Derivative;
ex:= \nabla{ A_{\mu} * B_{\nu} };
product_rule(_);
```

Property Derivative attached to \nabla{#}.

$\nabla\left(A_\mu B_\nu\right)$

$\nabla A_\mu B_\nu + A_\mu \nabla B_\nu$

This behaviour is a consequence of the fact that `Derivative` derives from `Distributable`. Note that the `Derivative` property does not automatically give you commuting derivatives, so that you can e.g. use it to write covariant derivatives. More specific derivative types exist too. An example are partial derivatives, declared using the `PartialDerivative` property. Partial derivatives are commuting and therefore automatically symmetric in their indices,

```
\partial{#}::PartialDerivative;
{a,b,m,n}::Indices(vector);
C_{m n}::Symmetric;
ex:=T^{b a} \partial_{a b}( C_{m n} D_{n m} );
```

Property PartialDerivative attached to \partial{#}.

Property Indices(position=free) attached to $[a,b,m,n]$.

Property Symmetric attached to C_{mn}.

$T^{ba}\partial_{ab}\left(C_{mn}D_{nm}\right)$

```
canonicalise(_);
```

$T^{ab}\partial_{ab}\left(C_{mn}D_{mn}\right)$

4 Manipulating expressions

4.1 Selecting parts of expressions

In many other computer algebra systems, you can select parts of results using the mouse, paste them into a new input cell, and then continue the computation. Naively this sounds like a nice feature to have, and it is indeed quite useful for quick computations. However, for larger projects, this feature quickly becomes a major source of trouble. Once you use the cut-n-paste technique, you are no longer able to make any changes in cells before the one with pasted content. Or rather, you can make changes, but they will not automatically propagate to into the pasted cell. Any effect of the change at the top of the notebook will have to be evaluated until the point of the cut-n-paste, and then you have to do the cut-n-paste again by hand.

Now this is fine if you just do a quick computation, as you will probably know precisely what you want to cut-n-paste. But if you give your notebook to someone else, this may no longer be clear. Worse, if you do not look at your notebook for some time, and then return after a few months (or years), you will most likely have forgotten completely what was the logic for the particular cut.

For this reason, Cadabra does not support cut-n-paste of output. But that does not mean that you cannot select parts of expressions for subsequent computation. For that, Cadabra has a more systematic logic, which is built around the `zoom` and `unzoom` commands.

4.1.1 Zooming into an expression

If you have a large expression, and want to select a part of it for further manipulation, while temporarily ignoring the rest, use the `zoom` command. It takes an expression and a pattern, and then suppresses all terms in the expression which do not match the pattern. An example:

```
ex:= \int{ c A + c**2 B + c D + c**2 A }{x};
```

$$\int \left(cA + c^2B + cD + c^2A\right)\ \mathrm{d}x$$

```
zoom(_, $c Q??$);
```

$$\int \left(cA + \ldots + cD + \ldots\right)\ \mathrm{d}x$$

This has selected all terms with a single factor of c, and suppressed the other ones (but keeping a reminder that those terms are still there, in the form of the dots). You can now work on the visible terms as usual, e.g. doing a substitution,

```
substitute(_, $A -> E$);
```

$$\int \left(cE + \ldots + cD + \ldots\right)\ \mathrm{d}x$$

In order to get back to the full expression, use `unzoom`,

```
unzoom(_);
```

$$\int \left(cE + c^2B + cD + c^2A\right)\ \mathrm{d}x$$

As you can see, the substitution has only changed the terms which were visible at the time.

4.2 Using multiple files and notebooks

At some point, you will encounter computations which are best separated out into their own notebook. Or you will do a computation which takes a long time, and you want to write an intermediate result into a file so that you can read it back later easily. There are two options for this in Cadabra: importing notebooks into other notebooks, or writing individual expressions to a file and reading them back.

4.2.1 Importing a notebook into another one

The simplest way to separate functionality is to simply write a separate notebook with the properties and expressions which you want to re-use elsewhere. In this way, writing a 'package' for Cadabra is nothing else but writing a separate notebook. You can import any notebook into another one by using the standard Python import logic.

Example

Let us say we have a notebook `library.cnb`, which contains a single cell with the following content:

```
{m,n,p,q,r}::Indices;
ex:=A_{q r} A_{q r};
```

You can now import this into another notebook by simply using

```
from library import *
```

Cadabra looks for the `library.cnb` notebook in your `PYTHONPATH` (just as in ordinary Python programs), as well as in the current directory. You can see that this worked by e.g. the following:

```
ex;
rename_dummies(ex);
```

$$A_{qr}A_{qr}$$

$$A_{mn}A_{mn}$$

Note that the import has thus not only imported the `ex` expression, but also the property information about the index set, which enabled the `rename_dummies` to work. Behind the scenes, what happens is that the import statement looks for a file `library.cnb`. If it finds this, it will first convert that file to a proper Python file (remember the `library.cnb` file is a Cadabra notebook, not a Python file). It then uses the standard Python logic to do the import.

4.2.2 Writing expressions to a file and reading them back

A somewhat more difficult way to re-use expressions is to write them to a file using standard Python methods, and then read them back elsewhere. This method is best used for long computations of which you want to write an intermediate result out to disk, to be read in later (instead of doing a re-computation). Be aware that if you write an expression to disk, you do not write the property information of any of the symbols in that expression to disk.

Example

The following example declares two expressions and writes them to disk. It then reads the expressions back in again and assigns them to different Python names.

```
ex1:= A_{m n} \sin{x};
ex2:= B_{m n};
with open("output.cdb", "w") as file:
   file.write( ex1.input_form()+"\n" )
   file.write( ex2.input_form()+"\n" )
```

$A_{mn} \sin x$

B_{mn}

```
with open("output.cdb", "r") as file:
   ex3=Ex( file.readline() )
   ex4=Ex( file.readline() )
```

```
ex3;
ex4;
```

$A_{mn} \sin x$

B_{mn}

Note that when written in this way, the file `output.cdb` only contains the expressions, not their names (`ex1` and `ex2` in the example above). Cadabra's expressions can also be written to disk using Python's pickle functionality. This makes the code slightly less messy, but note that the file will no longer be human-readable. If you use the pickle module, the example above would read:

```
import pickle

ex1:= A_{m n} \sin{x};
ex2:= B_{m n};
with open("output.pkl", "wb") as file:
   pickle.dump(ex1, file)
   pickle.dump(ex2, file)
```

$A_{mn} \sin x$

B_{mn}

```
with open("output.pkl", "rb") as file:
   ex3=pickle.load(file)
   ex4=pickle.load(file)
```

```
ex3;
ex4;
```

$A_{mn} \sin x$

B_{mn}

4.3 Default simplification

By default, Cadabra does very few things "by itself" with your expressions. It only collects equal terms, but even that can be turned off if you want to. The logic is that all simplification steps are contained in a function `post_process`, which is executed on every new input and on every completion of an algorithm. It can contain arbitrary code, but by default it reads

```
def post_process(ex):
    collect_terms(ex)
```

which simply collects equal terms. You can for instance apply a substitution on every input automatically,

```
def post_process(ex):
    distribute(ex)
    substitute(ex, $A_{m n} -> B_{m q} B_{q n}$)
    collect_terms(ex)
```

```
{m,n,p,q}::Indices(vector);
A_{m n}::Symmetric;
ex:=A_{m n} ( A_{n m} + C_{n m});
```

Attached property Indices(position=free) to $[m,\ n,\ p,\ q]$.

Attached property Symmetric to $B_{mq}B_{qn}$.

$B_{mq}B_{qn}B_{np}B_{pm} + B_{mq}B_{qn}C_{nm}$

As usual dummy indices have been relabelled appropriately. The `post_process` function can be redefined on-the-fly in the middle of a notebook.

4.4 Patterns, conditionals and regular expressions

Patterns in Cadabra are quite a bit different from those in other computer algebra systems, because they are more tuned towards the pattern matching of objects common in tensorial expressions, rather than generic tree structures. Cadabra knows about three different pattern types: *name patterns* (for single names), *object patterns* (for things which include indices and arguments) and *dummy patterns* (things for which the name is irrelevant, like indices).

Name patterns are things which match a single name in an object, without indices or arguments. They are constructed by writing a single question mark behind the name, as in

```
ex:= Q + R;
substitute(_, $A? + B? -> 0$);
```

$Q + R$

0

which matches all sums with two terms, each of which is a single symbol without indices or arguments. If you want to match instead any object, with or without indices or arguments, use the double question mark instead. To see the difference more explicitly, compare the two substitute commands in the following example

```
ex:=A_{m n} + B_{m n};
substitute(_, $A? + B? -> 0$ );
substitute(_, $A?? + B?? -> 0$ );
```

$A_{mn} + B_{mn}$

$A_{mn} + B_{mn}$

0

Note that it does not make sense to add arguments or indices to object patterns; a construction of the type `A??_{\mu}(x)` is meaningless and will be flagged as an error.

There is a special handling of objects which are dummy objects. Objects of this type do not need the question mark, as their explicit name is never relevant. You can therefore write

```
ex:= A_{m n};
substitute(_, $A_{p q}->0$);
```

A_{mn}

0

to set all occurrances of the tensor A with two subscript indices to zero, regardless of the names of the indices (as you can see, this command sets A_{pq} to zero). When index sets are declared using the `Indices` property, these will be taken into account when matching. When replacing object wildcards with something else that involves these objects, use the question mark notation also on the right-hand side of the rule. For instance,

```
ex:= C + D + E + F;
substitute(_, $A? + B? -> A? A?$, repeat=True);
```

$C + D + E + F$

$CC + EE$

replaces every consecutive two terms in a sum by the square of the first term. The following example shows the difference between the presence or absence of question marks on the right-hand side:

```
ex:= C + D;
substitute(_, $A? + B? -> A?$);
```

C

```
ex:= C + D;
substitute(_, $A? + B? -> A A$);
```

AA

So be aware that the full pattern symbol needs to be used on the right-hand side (in contrast to many other computer algebra systems).

Note that you can also use patterns to remove or add indices or arguments, as in

```
{\mu, \nu, \rho, \sigma}::Indices(vector);
ex:= A_{\mu} B_{\nu} C_{\nu} D_{\mu};
substitute(_, $A?_{\rho} B?_{\rho} -> \dot{A?}{B?}$, repeat=True);
```

Attached property Indices(position=free) to $[\mu,\ \nu,\ \rho,\ \sigma]$.

$A_{\mu}B_{\nu}C_{\nu}D_{\mu}$

$A \cdot DB \cdot C$

4.4.1 Conditionals

In many algorithms, patterns can be supplemented by so-called conditionals. These are constraints on the objects that appear in the pattern. In the example below, the substitution is not carried out, as the rule applies only to patterns where the n and p indices are not equal,

```
ex:= A_{m n} B_{n q};
substitute(_, $ A_{m n} B_{p q} | n != p -> 0$);
```

$A_{mn}B_{nq}$

$A_{mn}B_{nq}$

$A_{mn}B_{nq}$

Without the conditional, the substitution does apply,

```
ex:= A_{m n} B_{n q};
substitute(_, $ A_{m n} B_{p q} -> 0$);
```

$A_{mn}B_{nq}$

0

Note that the conditional follows directly after the pattern, not after the full substitution rule. A way to think about this is that the conditional is part of the pattern, not of the rule. The reason why the conditional follows the full pattern, and not directly the symbol to which it relates, is clear from the example above: the conditional is a “global” constraint on the pattern, not a local one on a single index.

These conditions can be used to match names of objects using regular expressions. In the following example, the pattern `M?` will match against `C7`,

```
ex:= A + B3 + C7;
substitute(_, $A + M? + N? | \regex{M?}{"[A-Z]7"} -> \sin(M? N?)/N?$);
```

$$A + B3 + C7$$

$$\sin\left(C7B3\right) B3^{-1}$$

Without the condition, the first match of `M?` would be against `B3`,

```
ex:= A + B3 + C7;
substitute(_, $A + M? + N? -> \sin(M? N?)/N?$);
```

$$A + B3 + C7$$

$$\sin\left(B3C7\right) C7^{-1}$$

4.5 Numerical evaluation of expressions

Cadabra is primarily a symbolic computer algebra system, in the sense that it focuses on symbolic expressions, not the numerical value they take when all symbols in them are replaced with values. However, Cadabra does have functionality to evaluate expressions numerically as well, using either a call through SymPy, or using its own internal expression evaluator. We will here focus on the latter, as it is by far the fastest.

Let us start with a simple example to understand the basics. The following code creates a Cadabra expression containing just $\cos(x)$, and then numerically evaluates that expression for 100 values of x in the range $[0, 2\pi]$.

```
import numpy as np
import matplotlib.pyplot as plt
```

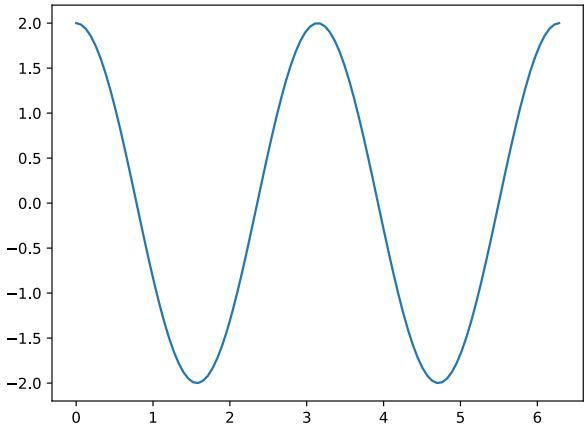
```
ex := 2\cos(2 x);
xv = np.linspace(0, np.pi*2, 100)
exv = nevaluate(ex, {$x$: xv} );
```

$$2\cos\left(2x\right)$$

```
<cadabra2.NTensor object at 0xffff79053a30>
```

The `nevaluate` function returns an `NTensor`, which is Cadabra's object to store numerical values of tensors. It can be converted to a numpy array by wrapping it in 'np.array', after which you can plot it:

```
plt.plot( xv, np.array(exv) );
```



To understand `nevaluate`, take a close look at the arguments. The first argument is the expression we want to evaluate. The second argument is a dictionary, in wich we list the symbols appearing in the expression (here just x) and the values that each such symbol takes (here it's the values in the array `xv`).

4.5.1 More complicated examples

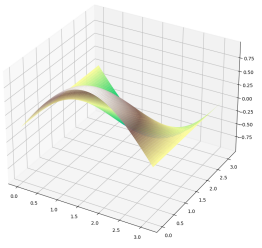
The example above evaluated a function of a single variable over a range of values of that variable. We can also use this to evaluate functions of multiple variables. The example below shows this.

```
ex:= \cos(x) \sin(y);
xv = np.linspace(0, np.pi, 100)
yv = np.linspace(0, np.pi, 100)
z = np.array( nevaluate(ex, {$x$: xv, $y$: yv} ) )
```

$$\cos x \sin y$$

The `z` variable is now a two-dimensional array, the first axis of which is the x-axis and the second the y-axis (more on this order below). We can plot such a data set by creating a meshgrid from the x and y values, and then feeding the lot into `plot_surface`:

```
XV, YV=np.meshgrid(xv, yv)
plt.figure(figsize=(20, 10))
ax = plt.axes(projection='3d')
plt.figure().subplots_adjust(top=1, bottom=0, left=0, right=1, wspace=0)
ax.plot_surface(XV, YV, z, rstride=1, cstride=1, cmap='terrain', edgecolor=None
    );
```



The order in which the axis of the result of `nevaluate` should be interpreted is determined by the order in which you list them in the values dictionary. Compare the following:

```
z1 = np.array( nevaluate(ex, {$x$: xv, $y$: yv} ))
z2 = np.array( nevaluate(ex, {$y$: yv, $x$: xv} ))
```

```
z1[10,20];
z2[10,20];
```

```
0.5633046988744114
```

```
0.2512712531759243
```

4.5.2 Supported elementary functions

At present the `nevaluate` function supports expressions with the following building blocks: multiplication/division, addition/subtraction, trigonometric functions, hyperbolic trigonometric functions, logarithms, exponential, square root.

4.6 Dynamic cell updates

By default the output generated by `display` will generate a new output cell. However, it is possible to put the output into an existing cell. In order to do this, use the `cell_id` parameter of the `display` call, passing it the ID of the cell you want to use for output. To obtain that cell, note that all `display` calls return the ID of the cell generated or reused.

The following example shows how to use this. On the first iteration of the loop we pass the ID '0', which leads to generation of a new cell. The call returns the ID of the newly generated cell, which is then used for any subsequent iterations of the loop.

```
from time import time, sleep
```

```
out=0
for i in range(100):
   if i!=0:
      sleep(0.02)
   out=display(i+1, cell_id=out)
```

```
100
```

You do not necessarily have to refer to an output cell generated by the current input cell, as the following example shows. The first input cell generates output, which is then changed by the following output cell.

```
out=0
for i in range(40):
   out=display("*"*i, cell_id=out)
   sleep(0.01)
```

```
*
```

```
for i in range(40):
   out=display("*"*(40-i), cell_id=out)
   sleep(0.01)
```

Note that while the updates are reasonably fast, there is of course some overhead; the following cell is a modified version of the first example, timing the total run.

```
start=time()
out=0
for i in range(1000):
   if i!=0:
      sleep(0.001)
   out=display(i+1, cell_id=out)
end=time()
display(f"time␣taken␣{(end-start):.2f}␣sec")
```

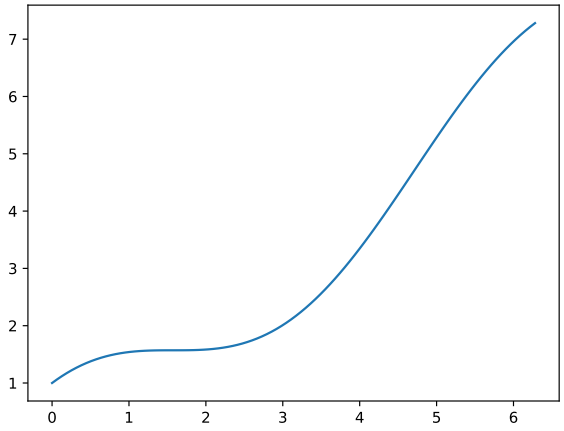
```
1000

time taken 1.35 sec
```

You can also use this mechanism to generate simple animations of functions. If you use `display` to put a plot into an existing output cell, the plot will be replaced.

```
from cdb.graphics.plot import plot
import numpy as np
```

```
out=0
for i in range(3):
   for v in np.concatenate([np.linspace(1.0, 3.0, 50), np.linspace(3.0, 1.0, 50)
       ]):
      ex := \cos(v x) + x;
      substitute(ex, $v -> @(v)$)
      out = display(plot(ex, ($x$, 0, 6.28)), out)
```


7
6
5
4
3
2
1
0
1
2
3
4
5
6

5 Writing your own packages

5.1 Programming in Cadabra

Cadabra is fully programmable in Python. At the most basic level this means that you can make functions which combine various Cadabra algorithms together, or write loops which repeat certain Cadabra algorithms. At a more advanced level, you can inspect the expression tree and manipulate individual subexpressions, or construct expressions from elementary building blocks.

5.1.1 Fundamental Cadabra objects: Ex and ExNode

The two fundamental Cadabra objects are the `Ex` and the `ExNode`. An object of type `Ex` represents a mathematical expression, and is what is generated if you type a line containing `:=`, as in

```
ex:=A+B;
type(ex);
```

$$A + B$$

```
<class 'cadabra2.Ex'>
```

An object of type `ExNode` is best thought of as an iterator. It can be used to walk an expression tree, and modify it in place (which is somewhat different from normal Python iterators; a point we will return to shortly). The most trivial way to get an iterator is to call the `top` member of an `Ex` object; think of this as returning a pointer to the topmost node of an expression,

```
ex.top();
type(ex.top());
```

```
A + B
```

```
<class 'cadabra2.ExNode'>
```

You will also encounter ExNodes when you do a standard Python iteration over the elements of an Ex, as in

```
for n in ex:
   type(n);
   display(n)
```

```
<class 'cadabra2.ExNode'>
```

```
A + B
```

```
<class 'cadabra2.ExNode'>
```

```
A
```

```
<class 'cadabra2.ExNode'>
```

```
B
```

As you can see, this 'iterates' over the elements of the expression, but in a perhaps somewhat unexpected way. We will discuss this in more detail in the next section. Important to remember from the example above is that the 'pointers' to the individual elements of the expression are ExNode objects. There are various other ways to obtain such pointers, using various types of 'filtering', more on that below as well.

Once you have an ExNode pointing to a subexpression in an expression, you can query it further for details about that subexpression.

```
ex:= A_{m n};
for i in ex.top().free_indices():
   display(i)
```

$$A_{mn}$$

```
m
```

```
n
```

The example above shows how, starting from an iterator which points to the top of the expression, you can get a new iterator which can iterate over all free indices.

5.1.2 ExNode and Python iterators

Before we continue, we should make a comment on how ExNode objects relate to Python iterators. For many purposes, ExNode objects behave as you expect from Python iterators: they allow you to loop over nodes of an Ex expression, you can call `next(...)` on them, and so on. However, there are some slight differences, which have to do with the fact that Cadabra wants to give you access to the nodes of the original Ex, so that you can modify

this original Ex in place. Consider for instance this example with a Python list of integers, with standard iterators:

```
q=[1,2,3,4,5];
for element in q:
   element=0
q;
```

[1, 2, 3, 4, 5] [1, 2, 3, 4, 5] It still produces the original list at the end of the day, because each element is a *copy* of the element in the list. With ExNodes you can actually modify the original Ex, as this example shows:

```
ex:=A + B + C + D;
for element in ex.top().terms():
   element.replace($Q$)
ex;
```

$$A + B + C + D$$

$$Q + Q + Q + Q$$

In this case, element is not an Ex corresponding to each of the 5 terms, but rather an ExNode, which is more like a pointer into the Ex object. The replace member function allows you to replace the building blocks of the original ex expression.

If you want to get a proper Ex object (so a *copy* of the element in the expression over which you are iterating), more like what you would get if iteration over Cadabra's expressions was an ordinary Python iteration, then you can use ExNode.ex():

```
ex:= A + 2 B + 3 C + 4 D;
lst=[]
for element in ex.top().terms():
   lst.append( element.ex() )
ex;
lst[2];
```

$$A + 2B + 3C + 4D$$

$$A + 2B + 3C + 4D$$

$$3C$$

Here the list lst contains copies of the individual terms of the ex expression.

A good way to remember about this is to keep in mind that Cadabra tries its best to allow you to modify expressions *in-place*. The ExNode iterators provide that functionality.

5.1.3 Traversing the expression tree

The ExNode iterator can be instructed to traverse expressions in various ways. The most basic iterator is obtained by using standard Python iteration with a for loop,

```
ex:= A + B + C_{m} D^{m};
```

$$A + B + C_m D^m$$

```
for n in ex:
   print(str(n))
```

```
\begin{verbatim}A + B + C_{m} D^{m}
A
B
C_{m} D^{m}
C_{m}
m
D^{m}
m
\end{verbatim}
```

The iterator obtained in this way traverses the expression tree node by node, and when you ask it to print what it is pointing to, it prints the entire subtree of the node it is currently visiting. If you are only interested in the name of the node, not the entire expression below it, you can use the `.name` member of the iterator:

```
for n in ex:
   print(str(n.name))
```

```
\begin{verbatim}\sum
A
B
\prod
C
m
D
m
\end{verbatim}
```

Often, this kind of 'brute force' iteration over expression elements is not very useful. A more powerful iterator is obtained by asking for all nodes in the subtree which have a certain name. This can be the name of a tensor, or the name of a special node, such as a product or sum,

```
for n in ex["C"]:
   display(n)
```

```
C_{m}
```

```
for n in ex["\\prod"]:
   display(n)
```

```
C_{m} D^{m}
```

The above two examples used an iterator obtained directly from an `Ex` object. Various ways of obtaining iterators over special nodes can be obtained by using member functions of `ExNode` objects themselves. So one often uses a construction in which one first asks for an iterator to the top of an expression, and then requests from that iterator a new one which can iterate over various special nodes. The example below obtains an iterator over all top-level terms in an expression, and then loops over its values.

```
for n in ex.top().terms():
   display(n)
```

```
A
```

```
B
```

```
C_{m} D^{m}
```

Two special types of iterators are those which iterate only over all arguments or only over all indices of a sub-expression. These are discussed in the next section.

5.1.4 Arguments and indices

There are various ways to obtain iterators which iterate over all arguments or all indices of an expression. The following example, with a derivative acting on a product, prints the argument of the derivative as well as all free indices.

```
\nabla{#}::Derivative;
ex:= \nabla_{m}{ A^{n}_{p} V^{p} };
```

Attached property Derivative to $\nabla\#$.

$\nabla_{m}\left(A^{n}{}_{p}V^{p}\right)$

```
for nabla in ex[r'\nabla']:
   for arg in nabla.args():
      print(str(arg))
   for i in nabla.free_indices():
      print(str(i))
```

```
\begin{verbatim}A^{n}_{p} V^{p}
m
n
\end{verbatim}
```

5.1.5 Querying properties

Properties which you attach to patterns can be queried in Python, though the functionality is somewhat limited. In order to query a pattern for a particular property, use the property's name together with the `get` method. An example:

```
A_{m n}::AntiSymmetric.
p1 = AntiSymmetric.get($A_{m n}$)
```

```
p1;
```

Property AntiSymmetric attached to A_{mn}.

```
p2 = Symmetric.get($A_{m n}$)
p2;
```

```
None
```

Some properties, like 'Weight' have an associated value. You can access these with the appropriate member function, so for this particular example you would do

```
x::Weight(value=42, label=field);
Weight.get($x$, label="field").value("field");
```

Property Weight attached to x.

```
42
```

5.1.6 Expression pattern matching

If you want to check whether an expression matches a particular pattern, use the `match` function of the `Ex` object. By default this is rather strict, requiring that indices match not only their type but also their name.

```
{m, n, k, l}::Indices(vector).
{a, b, c, d}::Indices(spinor).
$A_{m n}$.matches($A_{k l}$);
```

```
True
```

```
$A_{m n}$.matches($A_{m n}$);
```

```
True
```

```
$A_{m n}$.matches($A_{k l}$);
```

```
True
```

```
$A_{m n}$.matches($A_{a b}$);
```

```
False
```

Wildcard symbols will match any symbol,

```
$A_{m? n?}$.matches($A_{k l}$);
$A_{m? n?}$.matches($A_{a b}$);
```

```
True
```

```
True
```

```
$A??$.matches($A_{k l}$);
```

```
True
```

5.1.7 Example: covariant derivatives

The following example shows how you might implement the expansion of a covariant derivative into partial derivatives and connection terms.

```
def expand_nabla(ex):
    for nabla in ex[r'\nabla']:
        nabla.name=r'\partial'
        dindex = nabla.indices().__next__()
        for arg in nabla.args():
            ret:=0;
            for index in arg.free_indices():
                t2:= @(arg);
                if index.parent_rel==sub:
                    t1:= -\Gamma^{p}_{@(dindex) @(index)};
                    t2[index]:= _{p};
                else:
                    t1:= \Gamma^{@(index)}_{@(dindex) p};
                    t2[index]:= ^{p};
                ret += Ex(str(nabla.multiplier)) * t1 * t2
            nabla += ret
    return ex
```

The sample expressions below show how this automatically takes care of not introducing connections for dummy indices, and how it automatically handles indices which are more complicated than single symbols.

```
\nabla{#}::Derivative;
ex:= 1/2 \nabla_{a}{ h^{b}_{c} };
expand_nabla(ex);
```

Attached property Derivative to $\nabla\#$.

$$\frac{1}{2}\nabla_{a}h^{b}{}_{c}$$

$$\frac{1}{2}\partial_{a}\left(h^{b}{}_{c}\right)+\frac{1}{2}\Gamma^{b}{}_{ap}h^{p}{}_{c}-\frac{1}{2}\Gamma^{p}{}_{ac}h^{b}{}_{p}$$

```
ex:= 1/4 \nabla_{a}{ v_{b} w^{b} };
expand_nabla(ex);
```

$$\frac{1}{4}\nabla_{a}\left(v_{b}w^{b}\right)$$

$$\frac{1}{4}\partial_{a}\left(v_{b}w^{b}\right)$$

```
ex:= \nabla_{\hat{a}}{ h_{b c} v^{c} };
```

```
expand_nabla(ex);
```

$$\nabla_{\hat{a}} \left(h_{bc} v^c \right)$$

$$\partial_{\hat{a}} \left(h_{bc} v^c \right) - \Gamma^p{}_{\hat{a}b} h_{pc} v^c$$

5.2 Using Cadabra directly from C++

It is possible to use the functionality of Cadabra directly from C++ code, without dealing with the Python layer on top of it, and without using the notebook interface. The library is called `cadabra2++` and building and installation instructions are provided in the project's `README`. Here we describe how to use this library.

5.2.1 Simple example

Basic use of the Cadabra C++ library consists of five parts: creating a kernel, inserting object properties into the kernel, defining expressions, acting with algorithms on those expressions, and displaying the result. A minimal example is as follows:

```
#include "cadabra2++.hh"
#include <iostream>

using namespace cadabra;
using namespace cadabra::cpplib;

int main() {
   Kernel k(true);
   inject_property<AntiCommuting>(k, "{A,B}");
   auto ex = "A␣B␣-␣B␣A"_ex(k);
   sort_product sp(k, *ex);
   sp.apply_generic();
   collect_terms(k, *ex);
   sp.apply_generic();
   std::cout << pprint(k, ex) << std::endl;
}
```

The output of this program is `2 A B`.

Most of the above should be fairly easy to understand for anyone who has worked with the Python interface to Cadabra before. Note how properties are attached to objects using the `inject_property` function call. This takes the kernel as argument, as well as a textual expression of what you would use in the Python interface.

Algorithms are C++ objects, which you need to instantiate, and then run explicitly by calling the `apply_generic` function. As in the Python version, algorithms act in-place (mostly), and the `ex` expression above thus changes as the code progresses.

Finally, the expression is printed by using the ‘pprint‘ function. This is necessary because printing requires information stored in the Cadabra kernel.

6 Algorithms

6.1 Substitution and variation

6.1.1 `distribute`

Distribute factors over sums.

Rewrite a product of sums as a sum of products, as in

$$a\,(b+c) \to a\,b + a\,c\,.$$

This would read

```
ex:=a (b+c);
distribute(_);
```

$a\,(b+c)$

$ab + ac$

The algorithm in fact works on all objects which carry the `Distributable` property,

```
Op{#}::Distributable;
ex:=Op(A+B);
distribute(_);
```

Attached property Distributable to $Op\,(\#)$.

$Op\,(A+B)$

$Op\,(A) + Op\,(B)$

The primary example of a property which inherits the `Distributable` property is `PartialDerivative`. The `distribute` algorithm thus also automatically writes out partial derivatives of sums as sums of partial derivatives,

```
\partial{#}::PartialDerivative;
ex:=\partial_{m}{A + B + C};
distribute(_);
```

Attached property PartialDerivative to $\partial\#$.

$\partial_m \left(A + B + C \right)$

$\partial_m A + \partial_m B + \partial_m C$

6.1.2 `product_rule`

Apply the Leibnitz rule to a derivative of a product

Apply the product rule or "Leibnitz identity" to an object which has the `Derivative` property, i.e.

```
D{#}::Derivative;
ex:=D(f g);
product_rule(_);
```

Attached property Derivative to $D\#$.

$D \left(fg \right)$

$Dfg + fDg$

This of course also works for derivatives which explicitly mention indices or components, as well as for multiple derivatives, as in the example below.

```
D{#}::Derivative.
ex:=D_{m n}(f g);
```

$D_{mn} \left(fg \right)$

```
product_rule(_);
```

$D_m \left(D_n f g + f D_n g \right)$

```
distribute(_);
```

$D_m \left(D_n f g \right) + D_m \left(f D_n g \right)$

```
product_rule(_);
```

$D_m D_n f g + D_n f D_m g + D_m f D_n g + f D_m D_n g$

6.1.3 substitute

Generic substitution algorithm.

Generic substitution algorithm. Takes a rule or a set of rules according to which an expression should be modified. If more than one rule is given, it tries each rule in turn, until the first working one is encountered, after which it continues with the next node.

```
ex:=G_{\mu \nu \rho} + F_{\mu \nu \rho};
substitute(_, $F_{\mu \nu \rho} -> A_{\mu \nu} B_{\rho}$ );
```

$G_{\mu\nu\rho} + F_{\mu\nu\rho}$

$G_{\mu\nu\rho} + A_{\mu\nu}B_{\rho}$

```
ex:= A_{\mu \nu} B_{\nu \rho} C_{\rho \sigma};
substitute(_, $A_{m n} C_{p q} -> D_{m q}$ );
```

$A_{\mu\nu}B_{\nu\rho}C_{\rho\sigma}$

$D_{\mu\sigma}B_{\nu\rho}$

This command takes full care of dummy index relabelling, as the following example shows:

```
{m,n,q,r,s,t,u}::Indices(vector).
ex:= a_{m} b_{n};
```

$a_m b_n$

```
substitute(_, $a_{q} -> c_{m n} d_{m n q}$ );
```

$c_{qr}d_{qrm}b_n$

By postfixing a name with a question mark, it becomes a pattern. You do not need this for indices (as the examples above show) but it is necessary for other types of function arguments.

```
ex:= \sin{ x }**2 + 3 + \cos{ x }**2;
substitute(ex, $\sin{A?}**2 + \cos{A?}**2 = 1$);
```

$\sin x^2 + 3 + \cos x^2$

4

Substitute can match sub-products and sub-sums, and you do not have to specify terms or factors in the order in which they appear,

```
ex:= A + B + C + D;
substitute(_, $A+C=Q$);
```

$A + B + C + D$

$Q + B + D$

```
ex:= A B C D;
substitute(_, $B D = Q$);
```

$ABCD$

AQC

However, you can request that the match is for the full sum or product,

```
ex:= A B C D + A B C D E F;
substitute(_, $A B C D = 1$, partial=False);
```

$ABCD + ABCDEF$

$1 + ABCDEF$

It will respect non-commuting objects and will not match if that would require moving non-commuting objects through each other,

```
{Q,R,S,T}::NonCommuting;
ex:= Q R S T;
substitute(_, $S Q = 1$);
```

Attached property NonCommuting to $[Q,\ R,\ S,\ T]$.

$QRST$

$QRST$

6.1.4 `vary`

Generic variation algorithm for functional derivatives.

Generic variation command. Takes a rule or a set of rules according to which the terms in a sum should be varied. In every term, apply the rule once to every factor.

```
ex:= A B + A C;

vary(_, $A -> \epsilon D, B -> \epsilon C, C -> \epsilon A - \epsilon B$ );
```

$AB + AC$

$\epsilon DB + A\epsilon C + \epsilon DC + A\left(\epsilon A - \epsilon B\right)$

It also works when acting on powers, in which case it will use the product rule to expand them.

```
ex:= A**3;
vary(_, $A -> \delta{A}$);
```

$$A^3$$

$$3A^2\delta\left(A\right)$$

This algorithm is thus mostly intended for variational derivatives.

```
\partial{#}::PartialDerivative;
ex:= -\int{\partial_{\mu}{\phi} \partial^{\mu}{\phi} + m**2 \phi**2}{x};
```

Attached property PartialDerivative to $\partial\#$.

$$-\int \left(\partial_{\mu}\phi\partial^{\mu}\phi + m^2\phi^2\right)\,\mathrm{d}x$$

```
vary(ex, $\phi -> \delta{\phi}$);
```

$$-\int \left(\partial_{\mu}\delta\left(\phi\right)\partial^{\mu}\phi + \partial_{\mu}\phi\partial^{\mu}\delta\left(\phi\right) + 2m^2\phi\delta\left(\phi\right)\right)\,\mathrm{d}x$$

```
integrate_by_parts(ex, $\delta{\phi}$);
```

$$-\int \left(-\delta\left(\phi\right)\partial_{\mu}{}^{\mu}\phi - \partial^{\mu}{}_{\mu}\phi\delta\left(\phi\right) + 2m^2\phi\delta\left(\phi\right)\right)\,\mathrm{d}x$$

```
canonicalise(_)
sort_product(_);
```

$$-\int \left(-2\delta\left(\phi\right)\partial^{\mu}{}_{\mu}\phi + 2\delta\left(\phi\right)\phi m^2\right)\,\mathrm{d}x$$

```
factor_out(_, $\delta{\phi}$);
```

$$-\int \delta\left(\phi\right)\left(-2\partial^{\mu}{}_{\mu}\phi + 2\phi m^2\right)\,\mathrm{d}x$$

6.1.5 `expand_power`

Expand powers into repeated products

Expand powers into repeated products, i.e. do the opposite of `collect_factors`. For example,

```
ex:=(A B)**3;
```

$$\left(AB\right)^3$$

```
expand_power(_);
```

$$ABABAB$$

```
sort_product(_);
```

$AAABBB$

```
collect_factors(_);
```

A^3B^3

This command automatically takes care of index relabelling when necessary, as in the following example

```
{m,n,p,q,r}::Indices(vector).
ex:= (A_m B_m)**3;
```

$(A_mB_m)^3$

```
expand_power(_);
```

$A_mB_mA_nB_nA_pB_p$

6.1.6 unwrap

Move objects out of derivatives, accents or exterior products.

Move objects out of `Derivatives`, `Accents` or exterior (wedge) products when they do not depend on these operators. The most basic example is Accents, which will get removed from objects which do not depend on them, as in the following example:

```
\hat{#}::Accent;
\hat{#}::Distributable;
B::Depends(\hat);

ex:=\hat{A+B+C};
```

Attached property Accent to $\widehat{\#}$.

Attached property Distributable to $\widehat{\#}$.

Attached property Depends to B.

$A \widehat{+B+} C$

```
distribute(_);
```

$\widehat{A} + \widehat{B} + \widehat{C}$

```
unwrap(_);
```

$\widehat{B}$

Derivatives will be set to zero if an object inside does not depend on it. All objects which are annihilated by the derivative operator are moved to the front (taking into account anti-commutativity if necessary),

```
\partial{#}::PartialDerivative;
{A,B,C,D}::AntiCommuting;
x::Coordinate;
{B,D}::Depends(\partial{#});
```

Attached property PartialDerivative to $\partial\#$.

Attached property AntiCommuting to $[A,\ B,\ C,\ D]$.

Attached property Coordinate to x.

Attached property Depends to $[B,\ D]$.

```
ex:=\partial_{x}{A B C D};
```

$$\partial_{x}\left(ABCD\right)$$

```
unwrap(_);
```

$$-AC\partial_{x}\left(BD\right)$$

Note that a product remains inside the derivative; to expand that use `product_rule`. Here is another example:

```
\del{#}::LaTeXForm("\partial").
\del{#}::Derivative;
X::Depends(\del{#});
ex:=\del{X*Y*Z};
```

Attached property Derivative to $\partial\#$.

Attached property Depends to X.

$$\partial\left(XYZ\right)$$

```
product_rule(_);
```

$$\partial XYZ+X\partial YZ+XY\partial Z$$

```
unwrap(_);
```

$$\partial XYZ$$

Note that all objects are by default constants for the action of `Derivative` operators. If you want objects to stay inside derivative operators you have to explicitly declare that they depend on the derivative operator or on the coordinate with respect to which you take a derivative.

The final case where `unwrap` acts is when exterior products contain factors which are scalars (or forms of degree zero). The following example shows this.

```
{f,g}::DifferentialForm(degree=0).
{V, W}::DifferentialForm(degree=1).
{V,g}::AntiCommuting;
foo := f V ^ W g;
```

Attached property AntiCommuting to $[V,\ g]$.

$$(fV) \wedge (Wg)$$

```
unwrap(_);
```

$$-fgV \wedge W$$

As this example shows, `unwrap` takes into account commutativity properties (hence the sign flip).

6.1.7 `integrate_by_parts`

Integrate by parts away from the indicated expression

Integrate by parts. This requires an expression with an object carrying a `Derivative` property. The algorithm should be given an expression that any derivatives should be integrated away from. An example makes this more clear:

```
\partial{#}::PartialDerivative;
ex:= \int{ \partial_{m}{ A } B C D }{x};
```

Attached property PartialDerivative to $\partial\#$.

$$\int \partial_m ABCD \,\mathrm{d}x$$

```
integrate_by_parts(_, $A$);
```

$$-\int A\partial_m (BCD) \,\mathrm{d}x$$

```
product_rule(_);
```

$$-\int A (\partial_m BCD + B\partial_m CD + BC\partial_m D) \,\mathrm{d}x$$

```
distribute(_);
```

$$-\int (A\partial_m BCD + AB\partial_m CD + ABC\partial_m D) \,\mathrm{d}x$$

Note that `integrate_by_parts` only does the formal manipulation of moving the derivative around. If you want to discard derivatives of objects which are constant, you need to use the `Depends` property to indicate on which coordinates or derivatives objects depend, and the `unwrap` algorithm to eliminate derivatives of constants, as in the following lines.

```
{B,D}::Depends(\partial);
```

Attached property Depends to (B, D).

```
unwrap(ex);
```

$$-\int (A\partial_m BCD + ABC\partial_m D)\,\mathrm{d}x$$

6.2 Metrics and bundles

6.2.1 `eliminate_kronecker`

Eliminate Kronecker delta symbols.

Eliminates Kronecker delta symbols by performing index contractions. Also replaces contracted Kronecker delta symbols with the range over which the index runs, if known.

```
\delta_{m n}::KroneckerDelta.
ex:=A_{m p} \delta_{p q} B_{q n};
eliminate_kronecker(_);
```

$A_{mp}\delta_{pq}B_{qn}$

$A_{mq}B_{qn}$

The index range is set as usual with `Integer`,

```
{m,n,p,q}::Integer(0..d-1).
\delta_{m n}::KroneckerDelta.
ex:=\delta_{p q} \delta_{p q};
eliminate_kronecker(_);
```

$\delta_{pq}\delta_{pq}$

d

In order to eliminate metric factors (i.e. to 'raise' and 'lower' indices) use the algorithm `eliminate_metric`.

6.2.2 `eliminate_metric`

Eliminate metrics by raising or lowering indices.

Eliminate metric and inverse metric objects by raising or lowering indices.

```
{m, n, p, q, r}::Indices(vector, position=fixed).
{m, n, p, q, r}::Integer(0..9).
g_{m n}::Metric.
g^{m n}::InverseMetric.
g_{m}^{n}::KroneckerDelta.
g^{m}_{n}::KroneckerDelta.
ex:=g_{m p} g^{p m};
eliminate_metric(_);
```

$g_{mp}g^{pm}$

$g^{p}{}_{p}$

```
eliminate_kronecker(_);
```

10

Related algorithms are `eliminate_kronecker` and `eliminate_vielbein`. It is sometimes useful to eliminate only those metrics which have two dummy indices (so as to avoid changing indices on non-metric factors), as in the following example:

```
{a,b,c,d,e,f}::Indices(position=fixed);
g_{a b}::Metric;
g^{a b}::InverseMetric;
ex:=X_{a} g^{a b} g_{b c} g^{c d} g_{d e} g^{e f};
eliminate_metric(ex, repeat=True, redundant=True);
```

Property Indices(position=fixed) attached to $[a, b, c, d, e, f]$.

Property Metric attached to g_{ab}.

Property TableauSymmetry attached to g^{ab}.

$X_{a}g^{ab}g_{bc}g^{cd}g_{de}g^{ef}$

$X_{e}g^{ef}$

Without the `redundant=True` option, this would have reduced the expression to X^{f}.

6.2.3 `eliminate_vielbein`

Eliminates vielbein objects.

Indices of one type can be converted to another type by using a vielbein or inverse vielbein object.

```
{ m, n, p }::Indices(flat).
{ \mu, \nu, \rho }::Indices(curved).
e^{m}_{\mu}::Vielbein.
ex:= H_{m n p} e^{m}_{\mu} e^{p}_{\rho};
eliminate_vielbein(_, repeat=True);
```

$$H_{mnp}e^{m}{}_{\mu}e^{p}{}_{\rho}$$

$$H_{\mu n\rho}$$

This is similar to `eliminate_metric`.

6.2.4 `einsteinify`

Raise or lower indices of pairs which are both upper or lower.

In an expression containing dummy indices at the same position (i.e. either both subscripts or both superscripts), raise or lower one of the indices.

```
ex:= A_{m} A_{m};
einsteinify(_);
```

$$A_{m}A_{m}$$

$$A^{m}A_{m}$$

```
ex:= A^{m} A^{m};
einsteinify(_);
```

$$A^{m}A^{m}$$

$$A_{m}A^{m}$$

If an additional argument is given to this algorithm, it instead inserts "inverse metric" objects, with the name as indicated by the additional argument.

```
{m,n}::Indices.
ex:= A_{m} A_{m};
einsteinify(_, $\eta$);
```

$$A_{m}A_{m}$$

$$A_{m}A_{n}\eta^{mn}$$

```
ex:= A^{m} A^{m};
einsteinify(_, $\eta$);
```

$$A^{m}A^{m}$$

$A_m A_n \eta^{mn}$

Note that the second form requires that there are enough dummy indices defined through the use of `Indices`.

6.2.5 `epsilon_to_delta`

Replace a product of two epsilon tensors with a generalised delta

Replace a product of two epsilon tensors with a generalised delta according to the expression

$$\epsilon^{r_1\cdots r_d}\epsilon_{s_1\cdots s_d} = \frac{1}{\sqrt{|g|}}\varepsilon^{r_1\cdots r_d}\sqrt{|g|}\varepsilon_{s_1\cdots s_d} = \mathrm{sign}(g)\, d!\, \delta^{r_1\cdots r_d}_{s_1\cdots s_d}\,, \tag{6.1}$$

where $\mathrm{sign}(g)$ denotes the signature of the metric g used to raise/lower the indices (see `EpsilonTensor` for conventions on the epsilon tensor). When the indices are not ocurring up/down as in this expression, and the index position is not free, metric objects will be generated instead.

Here is an example:

```
{a,b,c,d}::Indices.
{a,b,c,d}::Integer(1..3).
\delta{#}::KroneckerDelta.
\epsilon_{a b c}::EpsilonTensor(delta=\delta).
ex:=\epsilon_{a b c} \epsilon_{a b d};
```

$\epsilon_{abc}\epsilon_{abd}$

```
epsilon_to_delta(_);
```

$2\delta_{cd}$

Remember that if the result is a generalised delta, you can expand it in terms of normal deltas using `expand_delta`,

```
ex:=\epsilon_{a b c} \epsilon_{a d e};
epsilon_to_delta(_);
expand_delta(_);
```

$\epsilon_{abc}\epsilon_{ade}$

$2\delta_{bdce}$

$\delta_{bd}\delta_{ce} - \delta_{cd}\delta_{be}$

In order for this algorithm to work, you need to make sure that the epsilon symbols in your expression are declared as `EpsilonTensor` *and* that these declarations involve a specification of the `delta` and/or `metric` symbols. As you can see from this example, contracted indices inside the generalised delta are automatically eliminated, as the algorithm `reduce_gendelta` is called automatically; if you do not want this use the optional argument `reduce=False`.

```
ex:=\epsilon_{a b c} \epsilon_{a b d};
epsilon_to_delta(_, reduce=False);
```

$\epsilon_{abc}\epsilon_{abd}$

$6\delta_{aabbcd}$

Note that the results typically depend on the signature of the space-time, which can be introduced through the optional `metric` argument of the `EpsilonTensor` property. Compare the two examples below:

```
{a,b,c,d}::Indices.
{a,b,c,d}::Integer(1..3).
\delta{#}::KroneckerDelta.
\epsilon_{a b c}::EpsilonTensor(delta=\delta, metric=g_{a b}).

g_{a b}::Metric(signature=-1).
ex:=\epsilon_{a b c} \epsilon_{a b c};
```

$\epsilon_{abc}\epsilon_{abc}$

```
epsilon_to_delta(_);
```

-6

```
g_{a b}::Metric(signature=+1).
ex:=\epsilon_{a b c} \epsilon_{a b c};
epsilon_to_delta(_);
```

$\epsilon_{abc}\epsilon_{abc}$

6

Note that you need to specify the full symbol for the metric, including the indices, whereas the `Kronecker` delta argument only requires the name without the indices (because a contraction can generate generalised Kronecker delta symbols with any number of indices).

6.2.6 `expand_delta`

Expand generalised Kronecker delta symbols

In Cadabra the `KroneckerDelta` property indicates a generalised Kronecker delta symbol. In order to expand it into standard two-index Kronecker deltas, use `expand_delta`, as in the example below.

```
\delta{#}::KroneckerDelta;
```

Attached property KroneckerDelta to $\delta\left(\#\right)$.

```
ex:=\delta^{a}_{b}^{c}_{d};
```

$$\delta^{a}{}_{b}{}^{c}{}_{d}$$

```
expand_delta(_);
```

$$\frac{1}{2}\delta^{a}{}_{b}\delta^{c}{}_{d} - \frac{1}{2}\delta^{c}{}_{b}\delta^{a}{}_{d}$$

```
ex:=\delta^{a}_{m}^{l}_{n} \delta_{a}^{c}_{b}^{d};
```

$$\delta^{a}{}_{m}{}^{l}{}_{n}\delta_{a}{}^{c}{}_{b}{}^{d}$$

```
expand_delta(_);
distribute(_);
eliminate_kronecker(_);
canonicalise(_);
```

$$\left(\frac{1}{2}\delta^{a}{}_{m}\delta^{l}{}_{n} - \frac{1}{2}\delta^{l}{}_{m}\delta^{a}{}_{n}\right)\left(\frac{1}{2}\delta_{a}{}^{c}\delta_{b}{}^{d} - \frac{1}{2}\delta_{b}{}^{c}\delta_{a}{}^{d}\right)$$

$$\frac{1}{4}\delta^{a}{}_{m}\delta^{l}{}_{n}\delta_{a}{}^{c}\delta_{b}{}^{d} - \frac{1}{4}\delta^{a}{}_{m}\delta^{l}{}_{n}\delta_{b}{}^{c}\delta_{a}{}^{d} - \frac{1}{4}\delta^{l}{}_{m}\delta^{a}{}_{n}\delta_{a}{}^{c}\delta_{b}{}^{d} + \frac{1}{4}\delta^{l}{}_{m}\delta^{a}{}_{n}\delta_{b}{}^{c}\delta_{a}{}^{d}$$

$$\frac{1}{4}\delta^{l}{}_{n}\delta_{m}{}^{c}\delta_{b}{}^{d} - \frac{1}{4}\delta^{l}{}_{n}\delta_{b}{}^{c}\delta_{m}{}^{d} - \frac{1}{4}\delta^{l}{}_{m}\delta_{n}{}^{c}\delta_{b}{}^{d} + \frac{1}{4}\delta^{l}{}_{m}\delta_{b}{}^{c}\delta_{n}{}^{d}$$

$$\frac{1}{4}\delta_{b}{}^{d}\delta^{c}{}_{m}\delta^{l}{}_{n} - \frac{1}{4}\delta_{b}{}^{c}\delta^{d}{}_{m}\delta^{l}{}_{n} - \frac{1}{4}\delta_{b}{}^{d}\delta^{c}{}_{n}\delta^{l}{}_{m} + \frac{1}{4}\delta_{b}{}^{c}\delta^{d}{}_{n}\delta^{l}{}_{m}$$

Note that it is in principle possible to get a result similar to the expanded form by using the Young projector and then canonicalising, but this is more expensive:

```
ex:=\delta^{a}_{b}^{c}_{d};
```

$$\delta^{a}{}_{b}{}^{c}{}_{d}$$

```
young_project_tensor(_);
```

$$\delta^{a}{}_{b}{}^{c}{}_{d}$$

6.2.7 `reduce_delta`

Simplify a self-contracted generalised delta.

Reduce a self-contracted generalised Kronecker delta symbol to a simpler expression without self-contractions, according to

$$n!\,\delta^{a_1\cdots a_n}_{b_1\cdots b_n}\,\delta^{b_1}_{a_1}\cdots\delta^{b_m}_{a_m} = \Big[\prod_{i=1}^{m}\big(d-(n-i)\big)\Big]\,(n-m)!\,\delta^{a_{m+1}\cdots a_n}_{b_{m+1}\cdots b_n}\,. \tag{6.2}$$

Here is an example:

```
\delta{#}::KroneckerDelta;
{m,n,q}::Integer(0..3);
ex:=\delta_{m}^{n}_{n}^{q};
```

Attached property KroneckerDelta to $\delta\left(\#\right)$.

Attached property Integer to $\left(m,n,q\right)$.

$\delta_m{}^n{}_n{}^q$

```
reduce_delta(_);
```

$$-\frac{3}{2}\delta_m{}^q$$

Note that this requires that the indices on the Kronecker delta symbol also carry an `Integer` property to specify their range.

6.3 Index manipulations

6.3.1 `combine`

Combine two consecutive indexbracket objects

Combine two consecutive objects with indexbrackets and consecutive contracted indices into one object with an indexbracket. An example with two contracted matrices:

```
ex:=(\Gamma_r)_{\alpha\beta} (\Gamma_{s t u})_{\beta\gamma};
combine(_);
```

$\left(\Gamma_r\right)_{\alpha\beta}\left(\Gamma_{stu}\right)_{\beta\gamma}$

$\left(\Gamma_r\Gamma_{stu}\right)_{\alpha\gamma}$

An example with a matrix and a vector:

```
ex:=(\Gamma_r)_{\alpha\beta} v_{\beta};
combine(_);
```

$$(\Gamma_r)_{\alpha\beta} v_{\beta}$$

$$(\Gamma_r v)_{\alpha}$$

The inverse of `combine` is `expand`.

6.3.2 `explicit_indices`

Make indices explicit on an expression with implicit indices.

In Cadabra you can write expressions which are understood to have indices suppressed, in order to get a cleaner notation. This is often used for vector/matrix notation, or when dealing with spinors. In order to inform Cadabra about these implicit indices, you use the `ImplicitIndex` property (which is also necessary to prevent Cadabra from moving these objects through each other when sorting products into canonical form). The `explicit_indices` algorithm can then make these indices explicit, which can sometimes make them easier to work with, for example when doing substitutions. In the following example we define two sets of indices, and several objects which are assumed to have implicit indices.

```
{m,n,p}::Indices(spacetime, position=fixed);
{a,b,c,d,e,f,g,h}::Indices(spinor, position=fixed);
\sigma^{p}::ImplicitIndex(\sigma^{p a}_{b});
\psi::ImplicitIndex(\psi_{a});
\chi::ImplicitIndex(\chi^{a});
```

Attached property Indices(position=fixed) to $[m,\ n,\ p]$.

Attached property Indices(position=fixed) to $[a,\ b,\ c,\ d,\ e,\ f,\ g,\ h]$.

Attached property ImplicitIndex to σ^{p}.

Attached property ImplicitIndex to ψ.

Attached property ImplicitIndex to χ.

The following is a valid expression for a spinor bilinear,

```
ex:= \psi \sigma^{m} \sigma^{n} \chi;
```

$$\psi\sigma^{m}\sigma^{n}\chi$$

We can now make the indices explicit using

```
explicit_indices(ex);
```

$\psi_a \sigma^{ma}{}_b \sigma^{nb}{}_c \chi^c$

This also works when there are trace operators, as is illustrated in the following example.

```
Tr{#}::LaTeXForm("{\rm Tr}").
Tr{#}::Trace(indices=spinor);
ex:= Tr(\sigma^{m} \sigma^{n} + \sigma^{n} \sigma^{m});
```

Attached property Trace to $\mathrm{Tr}\left(\#\right)$.

$\mathrm{Tr}\left(\sigma^m \sigma^n + \sigma^n \sigma^m\right)$

```
explicit_indices(_);
```

$\sigma^{ma}{}_b \sigma^{nb}{}_a + \sigma^{na}{}_b \sigma^{mb}{}_a$

6.3.3 `lower_free_indices`

Make all free indices in an expression subscripts.

Free indicies (indices declared with the `Indices(position=free)` property) can appear as subscripts or superscripts, but sometimes it is useful to move them all into the same position.

```
{a,b,c}::Indices(name=A, position=free);
{m,n,p}::Indices(name=B, position=fixed);
ex:=A_{a b m} B^{a b m};
```

Attached property Indices(position=free) to $[a,\ b,\ c]$.

Attached property Indices(position=fixed) to $[m,\ n,\ p]$.

$A_{abm} B^{abm}$

```
lower_free_indices(_);
```

$A_{abm} B_{ab}{}^{m}$

The opposite of this is `raise_free_indices`, which moves all indices to be superscripts.

6.3.4 `raise_free_indices`

Make all free indices in an expression superscripts.

Free indicies (indices declared with the `Indices(position=free)` property) can appear as subscripts or superscripts, but sometimes it is useful to move them all into the same position.

```
{a,b,c}::Indices(name=A, position=free);
{m,n,p}::Indices(name=B, position=fixed);
ex:=A_{a b m} B^{a b m};
```

Attached property Indices(position=free) to $[a,\ b,\ c]$.

Attached property Indices(position=fixed) to $[m,\ n,\ p]$.

$A_{abm}B^{abm}$

```
raise_free_indices(_);
```

$A^{ab}{}_{m}B^{abm}$

The opposite of this is `lower_free_indices`, which moves all free indices to be subscripts.

6.3.5 `split_index`

Split the range of an index into two subsets

Replace a sum by a sum-of-sums, abstractly. Concretely, replaces all index contractions of a given type by a sum of two terms, each with indices of a different type. Useful for Kaluza-Klein reductions and the like. An example makes this more clear:

```
{M,N,P,Q,R}::Indices(full).
{m,n,p,q,r}::Indices(space1).
{a,b,c,d,e}::Indices(space2).
```

```
ex:=A_{M p} B_{M p};
split_index(_, $M,m,a$);
```

$A_{Mp}B_{Mp}$

$A_{mp}B_{mp} + A_{ap}B_{ap}$

```
ex:=A_{M p} B_{M p};
split_index(_, $M,m,4$);
```

$A_{Mp}B_{Mp}$

$A_{mp}B_{mp} + A_{4p}B_{4p}$

Note that the two index types into which the original indices should be split can be either symbolic (as in the first case above) or numeric (as in the second case).

6.3.6 untrace

Take objects out of traces

When a trace contains objects which do not carry any implicit indices on which the trace acts, the untrace algorithm can be used to take them out of the trace. This is similar to the way in which unwrap takes objects out of derivatives when they do not depend on the object with respect to which the derivative is taken. Unless you declare objects to have a ImplicitIndex property, they will be taken out. The minimal example does not specify these indices, e.g.

```
{A,B}::ImplicitIndex.
tr{#}::Trace.
ex:= tr( q A B );
untrace(_);
```

$$tr\left(qAB\right)$$

$$qtr\left(AB\right)$$

In the declaration of a trace, it is possible to indicate over which indices the trace is being taken.

```
{a,b,c}::Indices(spinor).
{m,n,p}::Indices(vector).
C::ImplicitIndex(C_{a b}).
D::ImplicitIndex(D_{a b}).
E::ImplicitIndex(E^{m n}).
Tr{#}::Trace(indices=spinor).
```

```
ex:= Tr( C D E );
```

$$Tr\left(CDE\right)$$

```
untrace(_);
```

$$ETr\left(CD\right)$$

Note how, even though E has implicit indices, it has been moved out of the trace, as the latter is declared to be a trace over spinor indices, not vector indices.

6.3.7 rename_dummies

Rename dummy indices, within a set or from one set to another.

Rename the dummy indices in an expression. This can be necessary in order to make various terms in a sum use the same names for the indices, so that they can be collected.

```
{m,n,p,q,r,s}::Indices(vector);
ex:=A_{m n} B_{m n} - A_{p q} B_{p q};
```

Attached property Indices(position=free) to (m, n, p, q, r, s).

$A_{mn}B_{mn} - A_{pq}B_{pq}$

Using `canonicalise` does nothing here,

```
canonicalise(_);
```

$A_{mn}B_{mn} - A_{pq}B_{pq}$

However, renaming indices does the trick,

```
rename_dummies(_);
```

0

Note that the indices need to have been declared as being part of an index list, using the `Indices` property.

The algorithm can also be used to rename dummies from one set to another one, e.g. to change index conventions (this is used in many of Cadabra's packages). Here is an example.

```
{m,n,p,q}::Indices("one");
{a,b,c,d}::Indices("two");
ex:= A_{m} A^{m} + B_{m} C^{m} + A_{n} A^{n} + Q_{c d} R^{d c};
```

Attached property Indices(position=free) to $[m,\ n,\ p,\ q]$.

Attached property Indices(position=free) to $[a,\ b,\ c,\ d]$.

$A_{m}A^{m} + B_{m}C^{m} + A_{n}A^{n} + Q_{cd}R^{dc}$

The above expression has indices in two different sets. We now rename the first set to the second,

```
rename_dummies(_, "one", "two");
```

$2A_{a}A^{a} + B_{a}C^{a} + Q_{cd}R^{dc}$

6.3.8 `rewrite_indices`

Rewrite indices by contracting with vielbein or metric.

Rewrite indices on an object by contracting it with a second object which contains indices of both the old and the new type (a vielbein, in other words, or a metric). A vielbein example is

```
{m,n,p}::Indices(flat).
{\mu,\nu,\rho}::Indices(curved).
ex:=T_{m n p};
rewrite_indices(_, $T_{\mu\nu\rho}$, $e_{\mu}^{n}$);
```

T_{mnp}

$T_{\mu\nu\rho}e^{\mu}{}_{m}e^{\nu}{}_{n}e^{\rho}{}_{p}$

If you want to raise or lower an index with a metric, this can also be done with as an index rewriting command, as the following example shows:

```
{\mu,\nu,\rho,\sigma,\lambda,\kappa}::Indices(curved, position=fixed).
ex:=H_{\mu \nu \rho};
rewrite_indices(_, $H^{\mu \nu \rho}$, $g_{\mu \nu}$);
```

$H_{\mu\nu\rho}$

$H^{\sigma\lambda\kappa}g_{\mu\sigma}g_{\nu\lambda}g_{\rho\kappa}$

As these examples show, the desired form of the tensor should be given as the first argument, and the conversion object (metric, vielbein) as the second object.

6.3.9 expand

Write out products of objects with implicit indices.

Write out products of matrices and vectors inside indexbrackets, inserting new dummy indices for the contraction. This requires that the objects inside the index bracket are properly declared to have `Matrix` or `ImplicitIndex` properties.

Here is an example with multiple matrices:

```
{a,b,c,d,e}::Indices;
{A,B,C,D}::Matrix;

ex:= (A B C D)_{a b};
```

Attached property Indices(position=free) to (a,b,c,d,e).

Attached property Matrix to (A,B,C,D).

$(ABCD)_{ab}$

```
expand(_);
```

$A_{ac}B_{cd}C_{de}D_{eb}$

Compare the above to the following example, in which one of the objects inside the bracket is no longer a matrix:

```
ex:= (A B Q D)_{a b};
```

$$(ABQD)_{ab}$$

```
expand(_);
```

$$A_{ac}B_{cd}QD_{db}$$

Finally, an example with matrices carrying additional labels, as well as a vector object:

```
{\alpha,\beta}::Indices;
\Gamma{#}::Matrix;
v::ImplicitIndex;
```

Attached property Indices(position=free) to (α,β).

Attached property Matrix to $\Gamma(\#)$.

Attached property ImplicitIndex to v.

```
ex:=(\Gamma_{r} v)_{\alpha};
```

$$(\Gamma_r v)_{\alpha}$$

```
expand(_);
```

$$(\Gamma_r)_{\alpha\beta}v_{\beta}$$

6.4 Tensor component values

6.4.1 `complete`

Complete a set of substitution rules to cover related objects.

Complete a set of substitution rules with additional rules based on the properties of the objects appearing in the rules.

This can for instance be used to generate rules for the inverse components of the metric given the rules for the metric components themselves, as in the example below.

Note that the argument itself gets modified (amended) with the additional rules.

```
{r,t}::Coordinate.
{m,n,p,q}::Indices(values={r,t}).
g_{m n}::Metric.
g^{m n}::InverseMetric.
```

```
rl:={ g_{t t} = r, g_{t r} = r**2/a, g_{r t} = r**2/a, g_{r r} = 1 };
```

$$(g_{tt} = r, g_{tr} = \frac{r^2}{a}, g_{rt} = \frac{r^2}{a}, g_{rr} = 1)$$

```
complete(rl, $g^{m n}$);
```

$$(g_{tt} = r, g_{tr} = \frac{r^2}{a}, g_{rt} = \frac{r^2}{a}, g_{rr} = 1, g^{rr} = 1 + \frac{r^4}{a^2\left(r - \frac{r^4}{a^2}\right)}, g^{rt} = -\frac{r^2}{a\left(r - \frac{r^4}{a^2}\right)}, g^{tr} = -\frac{r^2}{a\left(r - \frac{r^4}{a^2}\right)}, g^{tt} = \frac{1}{r - \frac{r^4}{a^2}})$$

Note that this uses SymPy behind the scenes to do the scalar algebra and matrix inversion.

6.4.2 `evaluate`

Evaluate components of a tensor expression.

Given an abstract tensor expression and a set of rules for the components of tensors occurring in this expression, evaluate the components of the full expression.

The minimal information needed for this to work is a declaration of the indices used, and a declaration of the values that those indices use:

```
{r,t}::Coordinate.
{m,n,p,s}::Indices(values={t,r}).
ex:= A_{n m} B_{m n p} ( C_{p s} + D_{s p} );
```

$$A_{nm}B_{mnp}\left(C_{ps} + D_{sp}\right)$$

The list of component values should be given just like the list of rules for the `substitute` algorithm, that is, as equalities

```
rl:= [ A_{r t} = 3, B_{t r t} = 2, B_{t r r} = 5, C_{t r} = 1, D_{r t} = r**2*t
    , D_{t r}=t**2 ];
```

$$(A_{rt} = 3, B_{trt} = 2, B_{trr} = 5, C_{tr} = 1, D_{rt} = r^2t, D_{tr} = t^2)$$

The `evaluate` algorithm then works out the values of the components of the `ex` expression, which will be denoted with a $\square$ in its output,

```
evaluate(ex, rl);
```

$$\square_r = 6r^2t + 6$$

$$\square_t = 15t^2$$

6.5 Factorisation

6.5.1 `factor_in`

Collect terms in a sum that differ only by given pre-factors.

Given a list of symbols, this algorithm collects terms in a sum that only differ by pre-factors consisting of these given symbols. As an example,

```
ex:=a b + a c + a d;
```

$$ab + ac + ad$$

```
factor_in(_, $b,c$);
```

$$(b + c)\, a + ad$$

The name is perhaps most easily understood by thinking of it as a complement to `factor_out`. Or in case you are familiar with FORM, `factor_in` is like its `antibracket` statement.

The algorithm of course also works with indexed objects, as in

```
ex:=A_{m} B_{m} + C_{m} A_{m};
factor_in(_, $B_{n}, C_{n}$);
```

$$A_m B_m + C_m A_m$$

$$A_m B_m + C_m A_m$$

(not yet finished)

6.5.2 `factor_out`

Isolate common factors in a sum of products

Given a list of symbols, this algorithm tries to factor those symbols out of terms. As an example,

```
ex:= a b + a c e + a d;
```

$$ab + ace + ad$$

```
factor_out(_, $a$);
```

$$a\,(b + ce + d)$$

If you have non-commuting objects and want to factor out to the right, use the `right=True` option, as in

```
{A,B,C,D}::NonCommuting;
ex:= A B C D + B A C D;
factor_out(ex, $D$, right=True);
```

Attached property NonCommuting to $[A,\ B,\ C,\ D]$.

$$ABCD + BACD$$

$$(ABC + BAC)\,D$$

In case you are familiar with FORM, `factor_out` is like its `bracket` statement. If you add more factors to factor out, it works as in the following example.

```
ex:= a b + a c e + a c + c e + c d + a d;
```

$$ab + ace + ac + ce + cd + ad$$

```
factor_out(_, $a, c$);
```

$$a\,(b + d) + ac\,(e + 1) + c\,(e + d)$$

That is, separate terms will be generated for terms which differ by powers of the factors to be factored out.

The algorithm of course also works with indexed objects, as in

```
ex:= A_{m} B_{m} + C_{m} A_{m};
```

$$A_m B_m + C_m A_m$$

```
factor_out(_, $A_{m}$);
```

$$A_m\,(B_m + C_m)$$

6.6 Spinors and fermions

6.6.1 `expand_diracbar`

Simplify the Dirac bar of a composite object.

Rewrite the Dirac conjugate of a product of spinors and gamma matrices as a product of Dirac and hermitean conjugates. This uses

$$\overline{\psi} = i\psi^\dagger \Gamma^0\,, \tag{6.3}$$

together with

$$\Gamma_m^\dagger = \Gamma_0 \Gamma_m \Gamma_0\,. \tag{6.4}$$

For example,

```
\bar{#}::DiracBar.
\psi::Spinor(dimension=10).
\Gamma{#}::GammaMatrix.
ex:=\bar{\Gamma^{m n p} \psi};
```

$\overline{\Gamma^{mnp}\psi}$

```
expand_diracbar(_);
```

$\overline{\psi}\Gamma^{mnp}$

6.6.2 `fierz`

Perform a Fierz transformation on a product of four spinors

Change the order of the spinors in a four-spinor expression using a Fierz transformation. This relies on the generic fact that Dirac gamma matrices satisfy the completeness relation

$$\sum_a (\Gamma_a)_{ij} (\Gamma^a)_{kl} = \delta_{il}\delta_{jk} \,.$$

The following example explains the typical usage pattern.

```
{m,n,p,q,r,s}::Indices;
{m,n,p,q,r,s}::Integer(0..3);
\Gamma{#}::GammaMatrix;
\bar{#}::DiracBar;
{\theta, \lambda, \psi, \chi}::Spinor;
ex:=\bar{\theta} \Gamma_{m} \chi \bar{\psi} \Gamma^{m} \lambda;
```

Attached property Indices(position=free) to (m,n,p,q,r,s).

Attached property Integer to (m,n,p,q,r,s).

Attached property GammaMatrix to $\Gamma(\#)$.

Attached property DiracBar to $\overline{\#}$.

Attached property Spinor to $(\theta,\lambda,\psi,\chi)$.

$\overline{\theta}\Gamma_m\chi\overline{\psi}\Gamma^m\lambda$

```
fierz(_, $\theta, \lambda, \psi, \chi$);
```

$$-\frac{1}{4}\overline{\theta}\Gamma_m\Gamma^m\lambda\overline{\psi}\chi - \frac{1}{4}\overline{\theta}\Gamma_m\Gamma_n\Gamma^m\lambda\overline{\psi}\Gamma_n\chi - \frac{1}{8}\overline{\theta}\Gamma_m\Gamma_{np}\Gamma^m\lambda\overline{\psi}\Gamma_{pn}\chi$$

The argument to `fierz` is the required order of the fermions; note that this algorithm does not flip around Majorana spinors and `sort_spinors` should be used for that. Also note that it is important to define not only the symbols representing the spinors, Dirac bar and gamma matrices, but also the range of the indices.

6.6.3 `join_gamma`

Work out the product of two generalised Dirac gamma matrices.

Join two fully anti-symmetrised gamma matrix products according to the expression

$$\Gamma^{b_1\ldots b_n}\Gamma_{a_1\ldots a_m} = \sum_{p=0}^{\min(n,m)} \frac{n!m!}{(n-p)!(m-p)!p!}\Gamma^{[b_1\ldots b_{n-p}}{}_{[a_{p+1}\ldots a_m}\eta^{b_{n-p+1}\ldots b_n]}{}_{a_1\ldots a_{m-p}]}\,. \quad (6.5)$$

This is the opposite of `split_gamma`.

Without further arguments, the anti-symmetrisations will be worked out explicitly (changed from v1). The setting the flag "`expand`" to false instead keeps them implicit. Compare

```
\Gamma{#}::GammaMatrix(metric=g).
ex:= \Gamma_{m n} \Gamma_{p};
join_gamma(ex, expand=False);
```

$\Gamma_{mn}\Gamma_{p}$

$\Gamma_{mnp} + 2\Gamma_{m}g_{np}$

with

```
\Gamma{#}::GammaMatrix(metric=g).
ex:= \Gamma_{m n} \Gamma_{p};
join_gamma(ex, expand=True);
```

$\Gamma_{mn}\Gamma_{p}$

$\Gamma_{mnp} + \Gamma_{m}g_{np} - \Gamma_{n}g_{mp}$

Note that the gamma matrices need to have a metric associated to them in order for this algorithm to work. In order to reduce the number of terms somewhat, one can instruct the algorithm to make use of generalised Kronecker delta symbols in the result; these symbols are defined as

$$\delta^{r_1}{}_{s_1}{}^{r_2}{}_{s_2}\ldots{}^{r_n}{}_{s_n} = \delta^{[r_1}{}_{s_1}\delta^{r_2}{}_{s_2}\ldots{}^{r_n]}{}_{s_n}\,. \quad (6.6)$$

Anti-symmetrisation is implied in the set of even-numbered indices. The use of these symbols is triggered by the `use_gendelta` option,

```
{m,n,p,q}::Indices(position=fixed).
\Gamma{#}::GammaMatrix(metric=\delta).
ex:=\Gamma_{m n} \Gamma^{p q};
join_gamma(_, use_gendelta=True);
```

$$\Gamma_{mn}\Gamma^{pq}$$

$$\Gamma_{mn}{}^{pq} + \Gamma_m{}^q\delta_n{}^p - \Gamma_m{}^p\delta_n{}^q - \Gamma_n{}^q\delta_m{}^p + \Gamma_n{}^p\delta_m{}^q + 2\delta_n{}^p{}_m{}^q$$

6.6.4 `sort_spinors`

Sort Majorana spinor bilinears

Sorts Majorana spinor bilinears using the Majorana flip property, which for anti-commuting spinors takes the form

$$\overline{\psi}_1\Gamma_{r_1\cdots r_n}\psi_2 = \alpha\beta^n(-)^{\frac{1}{2}n(n-1)}\,\overline{\psi}_1\Gamma_{r_1\cdots r_n}\psi_2\,. \tag{6.7}$$

Here α and β determine the properties of the charge conjugation matrix,

$$\mathcal{C}^T = \alpha\mathcal{C}\,, \quad \mathcal{C}\Gamma_r\mathcal{C}^{-1} = \beta\Gamma_r^T\,. \tag{6.8}$$

Here is an example.

```
{\chi, \psi, \psi_{m}}::Spinor(dimension=10, type=MajoranaWeyl).
{\chi, \psi, \psi_{m}}::AntiCommuting.
\bar{#}::DiracBar.
\Gamma{#}::GammaMatrix.
{\psi_{m}, \psi, \chi}::SortOrder.
ex:=\bar{\chi} \Gamma_{m n} \psi;
```

$$\overline{\chi}\Gamma_{mn}\psi$$

```
sort_spinors(_);
```

$$-\overline{\psi}\Gamma_{mn}\chi$$

6.6.5 `split_gamma`

Split a Dirac gamma matrix off a generalised product of gamma matrices.

Given a generalised product of Dirac gamma matrices, rewrite it as a product with an explicit single gamma matrix. This is the inverse of the `join_gamma` algorithm. An example:

```
\Gamma{#}::GammaMatrix(metric=\eta);
ex:=\Gamma^{m n p};
```

Attached property GammaMatrix to $\Gamma(\#)$.

Γ^{mnp}

```
split_gamma(_, on_back=False);
```

$\Gamma^{m}\Gamma^{np} - \Gamma^{p}\eta^{mn} + \Gamma^{n}\eta^{mp}$

6.7 Sorting and canonicalisation

6.7.1 `asym`

Anti-symmetrise or symmetrise an expression in indicated indices or arguments

Anti-symmetrise or symmetrise (depending on the `antisymmetric` flag) a product or tensor in the indicated objects. This works both with normal objects and with indices. An example of the former:

```
ex:=A B C;
```

ABC

```
asym(_, $A,B,C$);
```

$$\frac{1}{6}ABC - \frac{1}{6}ACB - \frac{1}{6}BAC + \frac{1}{6}BCA + \frac{1}{6}CAB - \frac{1}{6}CBA$$

When used with indices, remember to also indicate whether you want to symmetrise upper or lower indices, as in the example below.

```
ex:=A_{m n} B_{p};
```

$A_{mn}B_{p}$

```
asym(_, $_{m}, _{n}, _{p}$);
```

$$\frac{1}{6}A_{mn}B_{p} - \frac{1}{6}A_{mp}B_{n} - \frac{1}{6}A_{nm}B_{p} + \frac{1}{6}A_{np}B_{m} + \frac{1}{6}A_{pm}B_{n} - \frac{1}{6}A_{pn}B_{m}$$

If you want to *symmetrise* in the indicated objects instead, use the `antisymmetric=False` flag:

```
ex:=A_{m n} B_{p};
asym(_, $_{m}, _{n}, _{p}$, antisymmetric=False);
```

$A_{mn}B_{p}$

$$\frac{1}{6}A_{mn}B_{p} + \frac{1}{6}A_{mp}B_{n} + \frac{1}{6}A_{nm}B_{p} + \frac{1}{6}A_{np}B_{m} + \frac{1}{6}A_{pm}B_{n} + \frac{1}{6}A_{pn}B_{m}$$

6.7.2 canonicalise

Bring a tensorial expression to canonical form by re-ordering indices.

Canonicalise a product of tensors, using the mono-term index symmetries of the individual tensors and the exchange symmetries of identical tensors. Tensor exchange takes into account commutativity properties of identical tensors.

Note that this algorithm does not take into account multi-term symmetries such as the Ricci identity of the Riemann tensor; those canonicalisation procedures require the use of young_project_tensor or young_project_product. Similarly, dimension-dependent identities are not taken into account, use decompose_product for those.

In order to specify symmetries of tensors you need to use symmetry properties such as Symmetric, AntiSymmetric or TableauSymmetry. The following example illustrates this.

```
A_{m n}::AntiSymmetric.
B_{p q}::Symmetric.
ex:=A_{m n} B_{m n};
canonicalise(_);
```

$$A_{mn}B_{mn}$$

$$0$$

If the various terms in an expression use different index names, you may need an additional call to rename_dummies before the terms get collected together:

```
{m,n,p,q,r,s}::Indices.
A_{m n}::AntiSymmetric.
C_{p q r}::AntiSymmetric.
ex:=A_{m n} C_{m n q} + A_{s r} C_{s q r};
canonicalise(_);
```

$$A_{mn}C_{mnq} + A_{sr}C_{sqr}$$

$$A_{mn}C_{qmn} - A_{rs}C_{qrs}$$

```
rename_dummies(_);
```

$$0$$

If you have symmetric or anti-symmetric tensors with many indices, it sometimes pays off to sort them to the end of the expression (this may speed up the canonicalisation process considerably).

6.7.3 young_project_product

Project all tensors in a product with their Young tableau projector.

Project all tensors in a product with their Young tableau projector. Each factor is projected in turn, after which the product is distributed and then canonicalised. This is often faster and more memory-efficient than first projecting every factor and then distributing.

Young projection can be used to find equalities between tensor polynomials which are due to multi-term symmetries, such as the Ricci identity in the example below.

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

ex:=2 R_{a b c d} R_{a c b d} - R_{a b c d} R_{a b c d};
```

$$2R_{abcd}R_{acbd} - R_{abcd}R_{abcd}$$

```
young_project_product(_);
```

$$0$$

6.7.4 young_project_tensor

Project tensors with their Young projector.

Project tensors with their Young projection operator. This works for simple symmetric or anti-symmetric objects, as in

```
A_{m n}::Symmetric.
ex:= A_{m n} A_{m p};
```

$$A_{mn}A_{mp}$$

```
young_project_tensor(_);
```

$$\left(\frac{1}{2}A_{mn} + \frac{1}{2}A_{nm}\right)\left(\frac{1}{2}A_{mp} + \frac{1}{2}A_{pm}\right)$$

but more generically works for any tensor which has a `TableauSymmetry` property attached to it.

```
A_{m n p}::TableauSymmetry(shape={2,1}, indices={0,2,1}).
ex:= A_{m n p};
```

$$A_{mnp}$$

```
young_project_tensor(_);
```

$$\frac{1}{3}A_{mnp} + \frac{1}{3}A_{pnm} - \frac{1}{3}A_{nmp} - \frac{1}{3}A_{pmn}$$

When the parameters `modulo_monoterm` is set to `True`, the resulting expression will be simplified using the monoterm symmetries of the tensor,

```
A_{m n p}::TableauSymmetry(shape={2,1}, indices={0,2,1}).
ex:= A_{m n p};
```

A_{mnp}

```
young_project_tensor(_, modulo_monoterm=True);
```

$$\frac{2}{3}A_{mnp} - \frac{1}{3}A_{npm} + \frac{1}{3}A_{mpn}$$

(in this example, the tensor is anti-symmetric in the indices 0 and 1, hence the simplification compared to the previous example).

6.7.5 `meld`

Combine terms when allowed by symmetries.

In a sum of terms, combine terms using mono-term and multi-term symmetries such that the expression does not use an overcomplete basis. The `meld` algorithm does *not* rewrite the expression to a canonical form, but it instead combines terms such that no terms remain which are a linear combination of the other terms. It can hence be used to prove equivalency of expressions under both mono-term and multi-term symmetries. A typical use cases where `meld` is preferable over e.g. `canonicalise` is when the expression contains tensors with multi-term symmetries:

```
R_{a b c d}::RiemannTensor;
ex:=R_{a b c d}R_{a b c d} + R_{a b c d}R_{a c b d};
meld(ex);
```

Attached property TableauSymmetry to R_{abcd}.

$R_{abcd}R_{abcd} + R_{abcd}R_{acbd}$

$$\frac{3}{2}R_{abcd}R_{abcd}$$

What has happened here is that the algorithm figured out that the first term is expressible in terms of the second, and has combined the two. If you write the terms in the opposite order, `meld` still combines them, but now in the form of the other term:

```
ex:=R_{a b c d}R_{a c b d}+ R_{a b c d}R_{a b c d};
meld(ex);
```

$R_{abcd}R_{acbd} + R_{abcd}R_{abcd}$

$3R_{abcd}R_{acbd}$

So `meld` does not canonicalise, but rather writes the expression such that there remain no linear dependencies between terms. This algorithm can of course be used for simpler situations, e.g. one which uses mono-term symmetries only:

```
A_{m n}::AntiSymmetric;
ex:=A_{m n} - A_{n m};
meld(ex);
```

Attached property AntiSymmetric to A_{mn}.

$A_{mn} - A_{nm}$

$2A_{mn}$

The `meld` algorithm can also be used as a quick way to collect terms which only differ by dummy index relabelling (even when there are no symmetries present), e.g.

```
ex:=Q_{m n} R^{m n} + R^{p q} Q_{p q};
```

$Q_{mn}R^{mn} + R^{pq}Q_{pq}$

```
meld(ex);
```

$2Q_{mn}R^{mn}$

The algorithm also handles cyclic symmetries of traces:

```
{\mu,\nu}::Indices(vector).
u^{\mu}::ImplicitIndex.
u^{\mu}::SelfNonCommuting.
tr{#}::Trace.
ex := tr{u^{\mu} u^{\mu} u^{\nu} u^{\nu}} -
     tr{u^{\mu} u^{\nu} u^{\nu} u^{\mu}};
meld(ex);
```

$tr\left(u^{\mu}u^{\mu}u^{\nu}u^{\nu}\right) - tr\left(u^{\mu}u^{\nu}u^{\nu}u^{\mu}\right)$

0

6.7.6 `sort_product`

Sort factors in a product

Sort factors in a product, taking into account any `SortOrder` properties. Also takes into account commutativity properties, such as `SelfCommuting`. If no sort order is given, it first does a lexographical sort based on the name of the factor, and if two names are identical,

does a sort based on the number of children and (if this number is equal) a lexographical comparison of the names of the children. Symbols starting with a backslash (greek letters etc.) get sorted to the right of roman letters.

The simplest sort is illustrated below,

```
ex := C B A D;
sort_product(_);
```

$$CBAD$$

$$ABCD$$

We can declare the objects to be anti-commuting, which then leads to

```
{A, B, C, D}::AntiCommuting.
ex := C B A D;
sort_product(_);
```

$$CBAD$$

$$-ABCD$$

For indexed objects, the anti-commutativity of components is indicated using the `SelfAntiCommuting` property,

```
\psi_{m}::SelfAntiCommuting.
ex := \psi_{n} \psi_{m} \psi_{p};
sort_product(_);
```

$$\psi_n \psi_m \psi_p$$

$$-\psi_m \psi_n \psi_p$$

Finally, the lexographical sort order can be overridden by using the `SortOrder` property,

```
{D, C, B, A}::SortOrder.
{A, B, C, D}::AntiCommuting.
ex := C B A D;
sort_product(_);
```

$$CBAD$$

$$-DCBA$$

6.7.7 `sort_sum`

Sort terms in a sum.

Sort terms in a sum, taking into account any `SortOrder` properties, or else sorting lexographically.

```
ex:=E+D+A+C+B;
```

$$E + D + A + C + B$$

```
sort_sum(_);
```

$$A + B + C + D + E$$

This is often useful in case sums appear as exponents; in this case it is necessary to first sort the sums before terms can be collected, as the following example shows.

```
ex:=a**(-1+d) - a**(d-1);
```

$$a^{(-1+d)} - a^{(d-1)}$$

```
sort_sum(_);
```

$$0$$

6.8 Weights and perturbations

6.8.1 `drop_weight`

Drop terms with given weight

Drop those terms for which a product has the indicated weight. Weights are computed by making use of the `Weight` property of symbols. This algorithm does the opposite of `keep_weight`.

As an example, consider the simple case in which we want to drop all terms with 3 fields. This is done using

```
{A,B}::Weight(label=field);
ex:=A B B + A A A + A B + B;
```

Attached property Weight to $[A,\ B]$.

$$ABB + AAA + AB + B$$

```
drop_weight(_, $field=3$);
```

$$AB + B$$

However, you can also do more complicated things by assigning non-unit weights to symbols, as in the example below,

```
{A,B}::Weight(label=field);
C::Weight(label=field, value=2);
ex:=A B B + A A A + A B + A C + B:
```

Attached property Weight to $[A,\ B]$.

Attached property Weight to C.

```
drop_weight(_, $field=3$);
```

$AB + B$

Weights can be "inherited" by operators by using the `WeightInherit` property. Here is an example using partial derivatives,

```
{\phi,\chi}::Weight(label=small, value=1);
\partial{#}::PartialDerivative;
\partial{#}::WeightInherit(label=all, type=multiplicative);
ex:=\phi \partial_{0}{\phi} + \partial_{0}{\lambda} + \lambda \partial_{3}{\chi
    };
```

Attached property Weight to $[\phi,\ \chi]$.

Attached property PartialDerivative to $\partial\#$.

Attached property WeightInherit to $\partial\#$.

$\phi\partial_0\phi + \partial_0\lambda + \lambda\partial_3\chi$

```
drop_weight(_, $small=1$);
```

$\phi\partial_0\phi + \partial_0\lambda$

6.8.2 `keep_weight`

Keep terms with indicated weight

Keep only those terms for which a product has the indicated weight. Weights are computed by making use of the `Weight` property of symbols. This algorithm does the opposite of `drop_weight`.

As an example, consider the simple case in which we want to keep all terms with 3 fields. This is done using

```
{A,B}::Weight(label=field);
ex:=A B B + A A A + A B + B;
keep_weight(_, $field=3$);
```

Attached property Weight to $[A,\ B]$.

$ABB + AAA + AB + B$

$$ABB + AAA$$

However, you can also do more complicated things by assigning non-unit weights to symbols, as in the example below,

```
{A,B}::Weight(label=field);
C::Weight(label=field, value=2);
ex:= A B B + A A A + A B + A C + B;
```

Attached property Weight to $[A,\ B]$.

Attached property Weight to C.

$$ABB + AAA + AB + AC + B$$

```
keep_weight(_, $field=3$);
```

$$ABB + AAA + AC$$

Weights also apply to tensorial expressions. Consider e.g. a situation in which we have a polynomial of the type

```
ex:=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};
```

$$c^{a} + c^{a}{}_{b}x^{b} + c^{a}{}_{bc}x^{b}x^{c} + c^{a}{}_{bcd}x^{b}x^{c}x^{d}$$

and we want to keep only the quadratic term. This can be done using

```
x^{a}::Weight(label=crd, value=1);
c^{#}::Weight(label=crd, value=0);
```

Attached property Weight to x^{a}.

Attached property Weight to $c^{\#}$.

```
keep_weight(ex, $crd=2$);
```

$$c^{a}{}_{bc}x^{b}x^{c}$$

Weights can be "inherited" by operators by using the `WeightInherit` property. Here is an example using partial derivatives,

```
{\phi,\chi}::Weight(label=small, value=1);
\partial{#}::PartialDerivative;
\partial{#}::WeightInherit(label=all, type=multiplicative);
ex:=\phi \partial_{0}{\phi} + \partial_{0}{\lambda} + \lambda \partial_{3}{\chi
    };
keep_weight(_, $small=1$);
```

Attached property Weight to $[\phi,\ \chi]$.

Attached property PartialDerivative to $\partial\#$.

Attached property WeightInherit to $\partial\#$.

$\phi\partial_0\phi + \partial_0\lambda + \lambda\partial_3\chi$

$\lambda\partial_3\chi$

If you want to use weights for dimension counting, in which operators can also carry a dimension themselves (e.g. derivatives), then use the `self` attribute,

```
reset();
```

```
{\phi,\chi}::Weight(label=length, value=1);

x::Coordinate;
\partial{#}::PartialDerivative;
\partial{#}::WeightInherit(label=length, type=multiplicative, self=-1);

ex:=\phi \partial_{x}{\phi} + \phi\chi + \partial_{x}{ \phi \chi**2 };
```

Attached property Weight to $[\phi,\ \chi]$.

Attached property Coordinate to x.

Attached property PartialDerivative to $\partial\#$.

Attached property WeightInherit to $\partial\#$.

$\phi\partial_x\phi + \phi\chi + \partial_x\left(\phi\chi^2\right)$

```
keep_weight(_, $length=1$);
```

$\phi\partial_x\phi$

6.9 Simplification

6.9.1 `collect_factors`

Collect identical factors in a product.

Collect factors in a product that differ only by their exponent. Note that factors containing sub- or superscripted indices do not get collected (i.e. A_mA^m does not get reduced to $(A_m)^2$).

```
ex:=A A B A B A;
```

$AABABA$

```
collect_factors(_);
```

A^4B^2

Arbitrary powers can be collected this way,

```
ex:=X X**(-1) X**(-4);
```

$XX^{-1}X^{-4}$

```
collect_factors(_);
```

X^{-4}

The exponent notation can be expanded again using `expand_power`.

```
ex:=X**4;
expand_power(_);
```

X^4

$XXXX$

6.9.2 `collect_terms`

Collect identical terms in a sum.

Collect terms in a sum that differ only by their numerical pre-factor. This is part of the default `post_process` function, so does not need to be called by hand.

Note that this command only collects terms which are identical, it does not collect terms which are different but mathematically equivalent. See `sort_sum` for an example.

6.9.3 `map_sympy`

Map Sympy algorithms to Cadabra expressions

Cadabra expressions are typically tensor expressions, which you cannot feed directly into Sympy. With the `map_sympy` function you can recursively apply Sympy algorithms to the scalar parts of Cadabra expressions.

The simplest example is when you have a scalar expression in Cadabra, for instance

```
ex:= \int{\sin(x)}{x};
```

$$\int \sin x \ \mathrm{d}x$$

```
map_sympy(ex);
```

$-\cos x$

The inert Cadabra expression gets evaluated by Sympy and then stored again in the ‘ex‘ object,

```
ex;
```

$$-\cos x$$

In more complicated cases you may have a tensorial expression which you would like to simplify using Sympy, for instance

```
ex:= (\sin(x)**2 + \cos(x)**2) A_{m} - A_{m};
```

$$\left((\sin x)^2 + (\cos x)^2\right) A_m - A_m$$

```
map_sympy(ex, "simplify");
```

$$0$$

6.9.4 `simplify`

Simplify the scalar part of an expression.

When expressions (or sub-expressions) involve scalars, simplification of such expressions can be ‘outsourced’ to an external scalar computer algebra system, at present either Sympy or Mathematica. The `simplify` algorithm finds all scalar sub-expressions and runs the simplification algorithm of one of these systems on them.

```
ex:= (\sin{x}**2 + \cos{x}**2) A_{m};
```

$$\left((\sin x)^2 + (\cos x)^2\right) A_m$$

```
simplify(_);
```

$$A_m$$

By default it will use the Sympy backend, but if you have compiled Cadabra on a system which has Mathematica installed, you can also switch it to use Mathematica instead, by using

```
kernel(scalar_backend="mathematica")
```

6.10 Representations

6.10.1 `decompose`

Decompose a tensor monomial on a given basis of monomials.

The basis should be given in the second argument. All tensor symmetries, including those implied by Young tableau Garnir symmetries, are taken into account. Example,

```
{m,n,p,q}::Indices(vector).
{m,n,p,q}::Integer(0..10).
R_{m n p q}::RiemannTensor.

ex:=R_{m n q p} R_{m p n q};
```

$R_{mnqp}R_{mpnq}$

```
decompose(ex, $R_{m n p q} R_{m n p q}$);
```

$$\left[-\frac{1}{2}\right]$$

Note that this algorithm does not yet take into account dimension-dependent identities, but it is nevertheless already required that the index range is specified.

6.10.2 decompose_product

Decompose a product of tensors by using Young projectors.

Decompose a product of tensors by writing it out in terms of irreducible Young tableau representations, discarding the ones which vanish in the indicated dimension, and putting the results back together again. This algorithm can thus be used to equate terms which are identical only in certain dimensions.

If there are no dimension-dependent identities playing a role in the product, then decompose_product returns the original expression,

```
{ m, n, p, q }::Indices(vector);
{ m, n, p, q }::Integer(1..4);
{ A_{m n p}, B_{m n p} }::AntiSymmetric;
ex:= A_{m n p} B_{m n q} - A_{m n q} B_{m n p};
```

Attached property Indices(position=free) to $\{m,n,p,q\}$.

Attached property Integer to $\{m,n,p,q\}$.

Attached property AntiSymmetric to $\{A_{mnp},B_{mnp}\}$.

$A_{mnp}B_{mnq}-A_{mnq}B_{mnp}$

```
decompose_product(_)
canonicalise(_);
```

$A_{pmn}B_{qmn}-A_{qmn}B_{pmn}$

However, in the present example, a Schouten identity makes the expression vanish identically in three dimensions,

```
{ m, n, p, q }::Integer(1..3);
ex:=A_{m n p} B_{m n q} - A_{m n q} B_{m n p};
decompose_product(ex)
canonicalise(ex);
```

Attached property Integer to $\{m, n, p, q\}$.

$$A_{mnp}B_{mnq} - A_{mnq}B_{mnp}$$

$$0$$

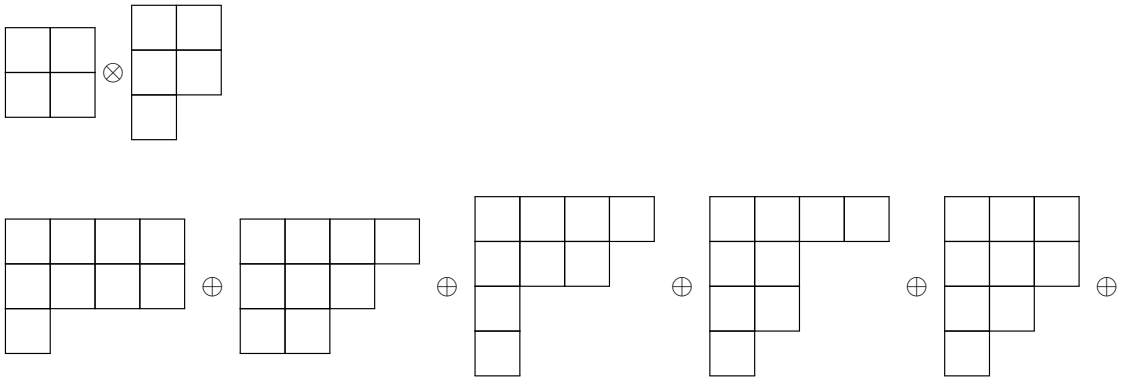
Note that `decompose_product` is unfortunately computationally expensive, and is therefore not practical for large dimensions.

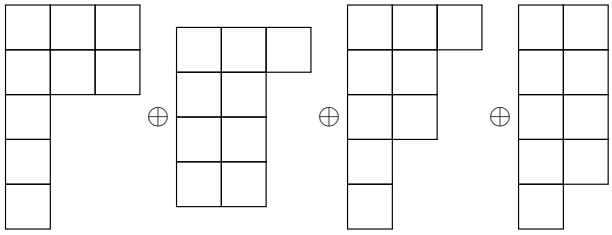
6.10.3 `lr_tensor`

Compute the tensor product of two Young tableaux

Compute the tensor product of two tableaux or filled tableaux. The algorithm acts on objects which have the `Tableau` or `FilledTableau` property, through which it is possible to set the dimension. The standard Littlewoord-Richardson algorithm is used to construct the tableaux in the tensor product. An example with `Tableau` objects is given below.

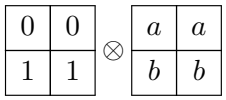
```
\tableau{#}::Tableau(dimension=10).
ex:=\tableau{2}{2} \tableau{2}{2}{1};
lr_tensor(_);
```



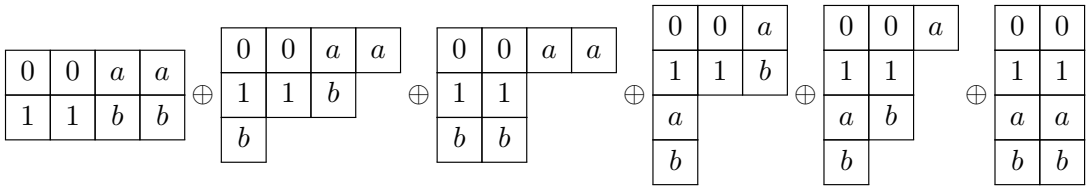


The same example, but now with `FilledTableau` objects, is

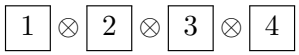
```
\ftableau{#}::FilledTableau(dimension=10).
ex:=\ftableau{0,0}{1,1} \ftableau{a,a}{b,b};
```



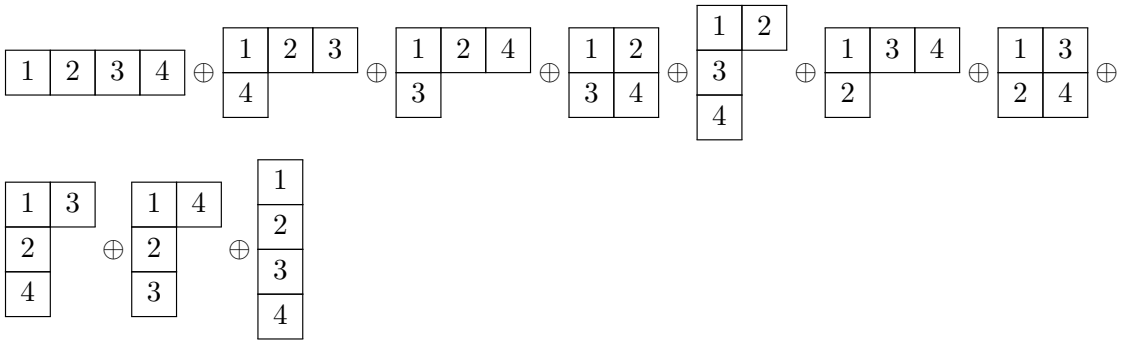
```
lr_tensor(_);
```



```
ex:=\ftableau{1} \ftableau{2} \ftableau{3} \ftableau{4};
```



```
converge(ex):
   lr_tensor(_)
   distribute(_)
;
```



6.11 Sub-expression manipulation

6.11.1 replace_match

Put the result of a sub-computation back into the original expression

This algorithm is the partner of `take_match`; see the documentation of that algorithm for further details.

6.11.2 take_match

Select a subset of terms in a sum for further computations.

The `take_match` and `replace_match` algorithms enable you to temporarily work only on a part of an expression. You select the terms that you want to work on with `take_match`. When you are done, put the result back into the larger expression with `replace_match`.

A simple example shows how this works:

```
ex:=A C + B D G + C D A;
```

$$AC + BDG + CDA$$

```
take_match(_, $D Q??$);
```

$$BDG + CDA$$

```
substitute(_, $C -> Q$);
```

$$BDG + QDA$$

```
replace_match(_);
```

$$AC + BDG + QDA$$

As you can see here, the replacement $C \to Q$ was only done on the 2nd and 3rd term of the original expression, to which we restricted all manipulations using the `take_match` command.

This of course also works with more complicated, tensorial expressions, as the example below shows.

```
ex:= A_{m n} \chi B^{m}_{p} + \psi A_{n p};
```

$$A_{mn}\chi B^{m}{}_{p} + \psi A_{np}$$

```
take_match(_, $\chi Q??$);
```

$$A_{mn}\chi B^{m}{}_{p}$$

```
substitute(_, $A_{m n} -> C_{m n}$);
```

$C_{mn}\chi B^{m}{}_{p}$

```
replace_match(_);
```

$C_{mn}\chi B^{m}{}_{p} + \psi A_{np}$

When you are working on a part of an expression, you can restrict attention further by applying `take_match` again. The `replace_match` then puts sub-expressions back into the larger expression in reverse order:

```
ex:=A B + C B + C D B;
```

$AB + CB + CDB$

```
take_match(_, $C Q??$);
```

$CB + CDB$

```
substitute(_, $C -> Q$);
```

$QB + QDB$

```
take_match(_, $D Q??$);
```

QDB

```
substitute(_, $B -> R$);
```

QDR

```
replace_match(_);
```

QB

```
replace_match(_);
```

$AB + QB$

6.11.3 `zoom`

Only show selected terms in a sum, and restrict subsequent algorithms to these terms.

Often you want manipulations to only apply to a selected subset of terms in a large sum. The `zoom` algorithm makes only certain terms visible, representing the remaining terms with dots. Any subsequent algorithms will only act on these visible terms.

Here is an expression with 5 terms,

```
ex:=\int{ A_{m n} + B_{m n} C + D_{m} F_{n} C + T_{m n} + B_{m n} R}{x};
```

$$\int \left(A_{mn} + B_{mn}C + D_{m}F_{n}C + T_{mn} + B_{mn}R\right)\ \mathrm{d}x$$

In order to restrict attention only to the terms containing a B_{mn} factor, we use

```
zoom(_, $B_{m n} Q??$);
```

$$\int (\ldots + B_{mn}C + \ldots + B_{mn}R)\ \mathrm{d}x$$

Subsequent algorithms only work on the visible terms above, not on the terms hidden inside the dots,

```
substitute(_, $C->Q$);
```

$$\int (\ldots + B_{mn}Q + \ldots + B_{mn}R)\ \mathrm{d}x$$

To make the hidden terms visible again, use `unzoom`, and note that the third term below has remained unaffected by the substitution above,

```
unzoom(_);
```

$$\int (A_{mn} + B_{mn}Q + D_mF_nC + T_{mn} + B_{mn}R)\ \mathrm{d}x$$

The `zoom/unzoom` combination is somewhat similar to the old deprecated `take_match/replace_match` algorithms, but makes it more clear that terms have been suppressed. It is possible to give `zoom` a list of patterns, in which case each term that is kept must match at least one of these patterns. See the examples below.

```
ex:= x A1 + x**2 A2 + y A3 + y**2 A4;
```

$$xA_1 + x^2A_2 + yA_3 + y^2A_4$$

```
zoom(ex, ${x A??, y A??}$);
```

$$xA_1 + \ldots + yA_3 + \ldots$$

```
unzoom(ex);
```

$$xA_1 + x^2A_2 + yA_3 + y^2A_4$$

```
zoom(ex, ${x A??, x**2 A??}$);
```

$$xA_1 + x^2A_2 + \ldots$$

6
Bibliography

[1] Kasper Peeters, Pierre Vanhove, and Anders Westerberg. "Supersymmetric higher-derivative actions in ten and eleven dimensions, the associated superalgebras and their formulation in superspace". In: *Class. Quant. Grav* 18 (2001), pp. 843–889. eprint: hep-th/0010167.

[2] Kasper Peeters and Anders Westerberg. "The Ramond-Ramond sector of string theory beyond leading order". In: *Class. Quant. Grav.* 21 (2004), pp. 1643–1666. eprint: hep-th/0307298.

[3] Kasper Peeters. "Introducing Cadabra: a symbolic computer algebra system for field theory problems". In: (2007). eprint: hep-th/0701238.

[4] Kasper Peeters. "Cadabra2: computer algebra for field theory revisited". In: *J. Open Source Softw.* 3.32 (2018), p. 1118. DOI: 10.21105/joss.01118.