Computer Physics Communications 185 (2014) 1719–1738

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

xTras: A field-theory inspired *xAct* package for mathematica[☆]

Teake Nutma [*]

Max-Planck-Institut für Gravitationsphysik (Albert Einstein Institut), Am Mühlenberg 1, 14476 Golm, Germany

ARTICLE INFO

Article history:
Received 23 September 2013
Received in revised form
27 January 2014
Accepted 3 February 2014
Available online 12 February 2014

Keywords:
Mathematica
Computer algebra
Tensor calculus
Field theory
Combinatorics
General relativity
Riemann tensor
Young tableaux

ABSTRACT

We present the tensor computer algebra package *xTras*, which provides functions and methods frequently needed when doing (classical) field theory. Amongst others, it can compute contractions, make Ansätze, and solve tensorial equations. It is built upon the tensor computer algebra system *xAct*, a collection of packages for Mathematica.

Program summary

Program title: xTras

Catalogue identifier: AESH_v1_0

Program summary URL: http://cpc.cs.qub.ac.uk/summaries/AESH_v1_0.html

Program obtainable from: CPC Program Library, Queen's University, Belfast, N. Ireland

Licensing provisions: GNU General Public License, version 3

No. of lines in distributed program, including test data, etc.: 155 879

No. of bytes in distributed program, including test data, etc.: 565 389

Distribution format: tar.gz

Programming language: Mathematica.

Computer: Any computer running Mathematica 6 or newer.

Operating system: Linux, Unix, Windows, OS X.

RAM: 100 Mb

Classification: 5.

External routines: xACT (www.xact.es)

Subprograms used:

Cat Id	Title	Reference
AEBH_v1_0	xPerm	CPC 179 (2008) 597
ADZK_v2_0	Invar Tensor Package 2.0	CPC 179 (2008) 586

Nature of problem:

Common problems in classical field theory: making Ansätze, computing contractions, solving tensorial equations, etc.

Solution method:

Various (group theory, brute-force, built-in Mathematica functions, etc.)

Running time:

1–60 s

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Tel.: +49 3315677114.
E-mail addresses: teake.nutma@gmail.com, teake.nutma@aei.mpg.de.

http://dx.doi.org/10.1016/j.cpc.2014.02.006
0010-4655/© 2014 Elsevier B.V. All rights reserved.

1. Introduction

xAct [1] is a free collection of powerful Mathematica packages for tensor computer algebra. Thanks to its implementation [2,3] of the Butler–Portugal algorithm [4–6], it can canonicalize tensor indices with respect to permutation symmetries extremely fast. On this solid basis a great number of applications have been built [7–13] that range from tensor spherical harmonics to perturbations around homogeneous cosmological backgrounds.

This paper describes the *xTras* package, one of these applications. *xTras* provides functions and methods that are frequently needed when doing (classical) field theory: computing contractions, making Ansätze, and solving equations, just to name a few. The package grew out of a need of the author for these particular functions, which were not present in any other *xAct* package.[1]

This paper is organized as follows. Section 2 describes how to install and run the package, Section 3 briefly reviews the basics usage of *xAct*, Section 4 demonstrates of capabilities of *xTras* with a couple of examples, and Section 5 contains function documentation. In addition to this paper, a complete list of all functions and their options can be found in either the built-in documentation of the package, or the online documentation at www.xact.es/xtras/documentation.

2. Installation

xTras can be installed by downloading the package from its website www.xact.es/xtras, unzipping it, and following the supplied instructions. Once installed, *xTras* can be loaded with the following command:

```
In := <<xAct`xTras`
      ------------------------------------------------------------
      Package xAct`xPerm` version 1.2.0, {2013,1,27}
      CopyRight (C) 2003-2013, Jose M. Martin-Garcia, under the General Public License.
      ------------------------------------------------------------
      Package xAct`xTensor` version 1.0.5, {2013,1,30}
      CopyRight (C) 2002-2013, Jose M. Martin-Garcia, under the General Public License.
      ------------------------------------------------------------
      Package xAct`xPert` version 1.0.3, {2013,1,27}
      CopyRight (C) 2005-2013, David Brizuela, Jose M. Martin-Garcia and Guillermo A. Mena Marugan, under the
      General Public License.
      ------------------------------------------------------------
      Package xAct`Invar` version 2.0.4, {2013,1,27}
      CopyRight (C) 2006-2013, J. M. Martin-Garcia, D. Yllanes and R. Portugal, under the General Public License.
      ------------------------------------------------------------
      Package xAct`xCoba` version 0.8.0, {2013,1,30}
      CopyRight (C) 2005-2013, David Yllanes and Jose M. Martin-Garcia, under the General Public License.
      ------------------------------------------------------------
      Package xAct`SymManipulator` version 0.8.5, {2013,4,13}
      CopyRight (C) 2011-2013, Thomas Bäckdahl, under the General Public License.
      ------------------------------------------------------------
      Package xAct`xTras` version 1.2.1, {2013,8,16}
      CopyRight (C) 2012-2013, Teake Nutma, under the General Public License.
      ------------------------------------------------------------
```

This loaded not only *xTras*, but also all other *xAct* packages that it depends on: *xPerm* [2], *xTensor* [3], *xPert* [7], *Invar* [9,10], *xCoba* [16], and *SymManipulator* [12]. Note that we have suppressed some print messages in the Mathematica output above, and have only shown the package info. In the rest of this paper, all print message will be suppressed.

Once *xTras* is loaded, the built-in documentation may be opened with the command

```
In := xTrasHelp[]                                                        1
```

or alternatively by first opening Mathematica's *Documentation Center* by pressing F1 and then searching for "xTras". Furthermore, information about *xTras* functions can be displayed, like all regular Mathematica functions, by typing `? functionname`. For example,

```
In := ? MakeTraceless                                                    2
```

MakeTraceless[*expr*] returns the traceless version of *expr*. »

gives a brief description of the function MakeTraceless. Pressing the » link opens its help page where more detailed documentation can be found.

3. xTensor basics

Before we discuss *xTras*, it is convenient to go over the basics of *xTensor* [3]. The *xTensor* package is more or less the cornerstone of *xAct*, as it implements the basic structures of manifolds, tensors, and Riemannian geometry (Table 1).

[1] Some of the functionality of *xTras* did however already exist in another computer algebra system, namely Cadabra [14,15].

Table 1
Basic commands in *xTensor*.

`DefManifold`[M, d, {$i_1, i_2, \ldots, i_n$}]	Defines the d-dimensional manifold M whose tensors will have indices $i_1, i_2, \ldots, i_n$.
`DefMetric`[*sign*, g [i_1, i_2], *cd*]	Defines a metric g of signature *sign* on the manifold of which i_1 and i_2 are indices, a covariant derivative *cd*, and all curvature tensor of g.
`DefTensor`[T [$i_1, \ldots, i_m$], M, *sym*]	Defines a tensor T with indices $i_1,\ldots,i_m$ and symmetry *sym* on the manifold M.
`ContractMetric`[*expr*]	Contracts all metrics in *expr*.
`ToCanonical`[*expr*]	Canonicalizes all tensors in *expr*.

The first step in any *xAct* calculation is always to define a manifold. This can be done with the aptly named command `DefManifold`:

```
In := DefManifold[M, 4, IndexRange[a,m]]                                    3
```

The first argument is the name of the manifold, in this case `M`. The second is its dimension. This either has to be an integer or a constant symbol (which needs to be defined as such with the command `DefConstantSymbol`). The last argument of `DefManifold` specifies the indices which will be used by tensors on the manifold; here `IndexRange[a,m]` is a convenient short-hand for `{a,b,c,d,e,f,g,h,i,j,k,l,m}`.

After defining a manifold, it is possible to define a metric on that manifold with the command `DefMetric`:

```
In := DefMetric[-1, metric[-a,-b], CD, PrintAs -> "g"]                      4
```

This defined a metric `metric` of signature −1 on the manifold `M`, because a and b are indices of `M`. Note that we could not use g for the name of the metric, because g is already an index. The option `PrintAs` makes sure that every time the metric appears in any output, it gets printed as g:

In := `metric[-a, -b]`
Out = $\mathrm{g_{ab}}$ 5

The minus signs in front of the indices indicate that they are covariant indices. Indices without a minus sign are contravariant:

In := `metric[a, b]`
Out = $\mathrm{g^{ab}}$ 6

Besides defining a metric, the command `DefMetric` also defined curvature tensors, like for instance the Riemann tensor,

In := `RiemannCD[-a, -b, -c, -d]`
Out = $\mathrm{R_{abcd}}$ 7

and the Ricci tensor:

In := `RicciCD[-a, -b]`
Out = $\mathrm{R_{ab}}$ 8

Their name indicates that they are associated to the covariant derivative `CD`, which also has been defined:

In := `CD[-a][RicciCD[-c, -d]]`
Out = $\nabla_{\mathrm{a}}\mathrm{R_{cd}}$ 9

By default, `DefMetric` defines a torsionless and metric compatible connection, and uses the conventions $[\nabla_a, \nabla_b]T_c = R_{abc}{}^dT_d$ and $R_{ab} = R_{acb}{}^c$ for the curvature tensors. Contractions of the Riemann tensor are automatically converted to Ricci tensors[2]:

In := `RiemannCD[-c, -b, -a, b]`
Out = $\mathrm{R_{ca}}$ 10

But contractions with an explicit metric are not converted:

In := `metric[b, d] RiemannCD[-c, -b, -a, -d]`
Out = $\mathrm{g^{bd}R_{cbad}}$ 11

[2] This behavior is actually controlled by the option `CurvatureRelations` of `DefMetric` (and `DefCovD`), which defaults to `True`. Torsion can be turned on by setting the option `Torsion` to `True`, and the relative signs for the Riemann and Ricci tensors are set via the global variables `$RiemannSign` and `$RicciSign`.

This is because *xTensor* does not automatically contract metrics. Contracting metrics can be done with the command `ContractMetric`, which does as its name suggests:

```
In := metric[a, c] RicciCD[-c, -b] // ContractMetric
Out = R^a_b
```
12

And indeed, applying `ContractMetric` to the previous example gives the Ricci tensor:

```
In := metric[b, d] RiemannCD[-c, -b, -a, -d] // ContractMetric
Out = R_ca
```
13

Note, however, that *xTensor* also does not automatically rewrite R_{ca} to R_{ac}, even though the Ricci tensor is symmetric. To achieve this, we have to use the function `ToCanonical`:

```
In := RicciCD[-c, -a] // ToCanonical
Out = R_ac
```
14

Very loosely speaking, `ToCanonical` tries to sort indices as much as possible based on the symmetries of the tensors in the expression (see [2] for more details). Needless to say, it also works on more complicated expressions:

```
In := RicciCD[c, d] RiemannCD[-d, -b, -a, -c] // ToCanonical
Out = -R^cd R_acbd
```
15

At the moment, `ToCanonical` only simplifies so-called *mono-term* symmetries, which are of the form $T_{i_1\cdots i_n} = \pm T_{\sigma(i_1\cdots i_n)}$, where $\sigma \in S_n$ is a permutation of the indices. It does not simplify so-called *multi-term* symmetries, which are of the form $T_{i_1\cdots i_n} = \pm T_{\sigma_1(i_1\cdots i_n)} \pm T_{\sigma_2(i_1\cdots i_n)} + \cdots$. One example of a multi-term symmetry is for instance the Bianchi identity $R_{[abc]d} = 0$.

After having covered the very basics of *xTensor*, we are now ready to tackle more advanced examples with the help of functions in *xTras*.

4. Examples

We will now demonstrate the features of *xTras*, or at least some of them, on the basis of two examples. The functions used here are described in more detail in Section 5.

4.1. Spin 2 on a flat background

In this section we will construct a gauge invariant theory of a free spin 2 field on a flat background. In doing so, we will recover the linearized Einstein equations (Table 2).

After loading the package, we have to define a manifold and a flat metric. This can be done as follows:

```
In := DefManifold[M, dim, IndexRange[a,m]]
In := DefMetric[
        -1, metric[-a,-b], PD, PrintAs -> "η",
        FlatMetric -> True, SymbolOfCovD -> {",","∂"}
      ]
```
16

This did not define a new covariant derivative, but instead set the pre-existing partial derivative PD to be metric compatible with `metric`. Furthermore, we need to tell the function `SymmetryOf` that the metric is constant:

```
In := SetOptions[SymmetryOf, ConstantMetric -> True]
```
17

Besides defining a manifold and a metric, we also need to define a symmetric spin two field and a gauge vector:

```
In := DefTensor[H[-a, -b], M, Symmetric[{-a, -b}], PrintAs -> "h"]
In := DefTensor[xi[a], M, PrintAs -> "ξ"]
```
18

We are now ready to begin the actual computation. We will construct all possible terms for the action, and make an Ansatz out of them. Because we are not interested in total derivatives, it suffices to consider terms of the form $h \cdot \partial \cdot \partial \cdot h$. First, find all of these terms:

```
In := Sterms = AllContractions[ H[a, b] PD[c]@PD[d]@H[e, f] ]
```
$$\text{Out} = \{h^{ab}\partial_b\partial_a h^c{}_c,\ h^{ab}\partial_c\partial_b h_a{}^c,\ h^a{}_a\partial_c\partial_b h^{bc},\ h^{ab}\partial_c\partial^c h_{ab},\ h^a{}_a\partial_c\partial^c h^b{}_b\}$$
19

Now construct the action:

```
In := S = MakeAnsatz[Sterms]
```
$$\text{Out} = C_1 h^{ab}\partial_b\partial_a h^c{}_c + C_2 h^{ab}\partial_c\partial_b h_a{}^c + C_3 h^a{}_a\partial_c\partial_b h^{bc} + C_4 h^{ab}\partial_c\partial^c h_{ab} + C_5 h^a{}_a\partial_c\partial^c h^b{}_b$$
20

Table 2
xTras functions used in Section 4.1. They are described in more detail in Section 5.

AllContractions[*expr*]	Computes all possible contractions of *expr*.
MakeAnsatz[{$e_1, e_2, \ldots$}]	Makes an Ansatz out of the list entries $e_1, e_2, \ldots$.
CollectTensors[*expr*]	Groups all tensorial terms in *expr* together.
SolveConstants[*expr*]	Attempts to solve the system *expr* of tensorial equations for all constant symbols appearing in *expr*.

Table 3
New *xTras* functions used in Section 4.2. They are described in more detail in Section 5.

EulerDensity[*cd*]	Gives the Euler density associated to the covariant derivative *cd*.
VarL[*g*[-*a*, −*b*]][*L*]	Computes $\frac{1}{\sqrt{\lvert g\rvert}}\frac{\delta\sqrt{\lvert g\rvert}L}{\delta g_{ab}}$.
FullSimplification[][*expr*]	Tries to simplify *expr* as much as possible, taking Bianchi identities into account and sorting covariant derivatives.
ConstructDDIs[*expr*]	Constructs all scalar dimensional dependent identities that can be build out of *expr*.
SolveTensors[*expr*]	Attempts to solve the system *expr* of tensorial equations for all tensors in *expr*.

The equations of motion are then:

```
In := eom = VarD[H[-a, -b], PD][S] // CollectTensors
```
Out = $(C_1 + C_3)\partial^b\partial^a h^c{}_c + C_2\partial_c\partial^a h^{bc} + C_2\partial_c\partial^b h^{ac} + 2C_4\partial_c\partial^c h^{ab} + (C_1 + C_3)\eta^{ab}\partial_d\partial_c h^{cd} + 2C_5\eta^{ab}\partial_d\partial^d h^c{}_c$ 21

We want to make the action and the equations of motion gauge invariant under the following gauge transformation of the spin two field:

```
In := δH = 2 Symmetrize[ PD[-a]@xi[-b] ]
```
Out = $\partial_a\xi_b + \partial_b\xi_a$ 22

To that end, we compute the gauge variation of the action δS to be

```
In := δS = δH eom // CollectTensors
```
Out = $2(C_1 + C_3)\partial_b\partial_a h^c{}_c\partial^b\xi^a + 2C_2\partial^b\xi^a\partial_c\partial_a h_b{}^c + 2C_2\partial^b\xi^a\partial_c\partial_b h_a{}^c$
$+2(C_1 + C_3)\partial_a\xi^a\partial_c\partial_b h^{bc} + 4C_4\partial^b\xi^a\partial_c\partial^c h_{ab} + 4C_5\partial_a\xi^a\partial_c\partial^c h^b{}_b$ 23

Up to total derivatives, this should be zero. We can eliminate total derivatives by removing all derivatives from the gauge parameter with the help of `VarD`:

```
In := δS = xi[a] VarD[xi[a], PD][δS] // CollectTensors
```
Out = $-2(C_1 + C_2 + C_3)\xi^a\partial_c\partial_b\partial_a h^{bc} - 2(C_1 + C_3 + 2C_5)\xi^a\partial_c\partial^c\partial_a h^b{}_b - 2(C_2 + 2C_4)\xi^a\partial_c\partial^c\partial_b h_a{}^b$ 24

Finally, we demand the above to be zero by solving for the unknown constants:

```
In := sols = SolveConstants[δS == 0]
```
Out = $\{\{C_3 \to -C_1 - C_2,\ C_4 \to -\frac{1}{2}C_2,\ C_5 \to \frac{1}{2}C_2\}\}$ 25

Plugging this solution into the action, we find

```
In := S /. First[sols]
```
Out = $C_1 h^{ab}\partial_b\partial_a h^c{}_c + C_2 h^{ab}\partial_c\partial_b h_a{}^c + (-C_1 - C_2)h^a{}_a\partial_c\partial_b h^{bc} - \frac{1}{2}C_2 h^{ab}\partial_c\partial^c h_{ab} + \frac{1}{2}C_2 h^a{}_a\partial_c\partial^c h^b{}_b$ 26

The coefficient C_2 parameterizes an overall normalization, and the coefficient C_1 a total derivative. Indeed, C_1 does not appear in the final equations of motion:

```
In := eom /. First[sols] /. C2 -> 1
```
Out = $-\partial^b\partial^a h^c{}_c + \partial_c\partial^a h^{bc} + \partial_c\partial^b h^{ac} - \partial_c\partial^c h^{ab} - \eta^{ab}\partial_d\partial_c h^{cd} + \eta^{ab}\partial_d\partial^d h^c{}_c$ 27

These are precisely the linearized Einstein equations.

4.2. Gauss–Bonnet term

In this section we will show that the Euler density in four dimensions, also known as the Gauss–Bonnet term, is topological. That is, we will show that its equations of motion vanish identically (Table 3).

We will begin from scratch, and define a manifold and metric:

```
In := DefManifold[M, 4, IndexRange[a,m]]
In := DefMetric[-1, metric[-a,-b], CD, PrintAs->"g"]
```
28

Next, we determine the Gauss–Bonnet term via the function EulerDensity:

```
In := GBterm = NoScalar @ EulerDensity[CD]
```
Out = $4R_{ab}R^{ab} - R^2 - R_{abcd}R^{abcd}$

29

The `NoScalar` call removed any `Scalar` heads in the expression (see also Section 5.4.3). Note that EulerDensity omits the overall factor $\sqrt{-g}$, so technically speaking `GBterm` is not a density. The equations of motion of the Gauss–Bonnet term can be determined with the function VarL, and simplified with FullSimplification:

```
In := eom = FullSimplification[] @ VarL[metric[-a, -b]] @ GBterm
```
Out = $-4R^{ac}R^{b}{}_{c} + 2g^{ab}R_{cd}R^{cd} + 2R^{ab}R - \frac{1}{2}g^{ab}R^2 - 4R^{cd}R^{a}{}_{c}{}^{b}{}_{d} + 2R^{acde}R^{b}{}_{cde} - \frac{1}{2}g^{ab}R_{cdef}R^{cdef}$

30

Because the Gauss–Bonnet term is topological, the above should identically be zero. There are no further simplifications coming from Bianchi identities that we can use: FullSimplification took care of most of them, and if there were some remaining we still could not use them to get rid of the Ricci tensors.

So the above equations of motion can only be zero due to dimensionally dependent identities. We can obtain the relevant identities with a call to ConstructDDIs:

```
In := ddis = ConstructDDIs[
         RiemannCD[a, b, c, d] RiemannCD[e, f, g, h],
         {a, b}
      ]
```
Out = $\{R^{ac}R^{b}{}_{c} - \frac{1}{2}g^{ab}R_{cd}R^{cd} - \frac{1}{2}R^{ab}R + \frac{1}{8}g^{ab}R^2 + R^{cd}R^{a}{}_{c}{}^{b}{}_{d} - R^{acde}R^{b}{}_{cde} + R^{acde}R^{b}{}_{dce} + \frac{1}{4}g^{ab}R_{cdef}R^{cdef} - \frac{1}{4}g^{ab}R_{cedf}R^{cdef},$

$R^{acde}R^{b}{}_{cde} - 2R^{acde}R^{b}{}_{dce} - \frac{1}{4}g^{ab}R_{cdef}R^{cdef} + \frac{1}{2}g^{ab}R_{cedf}R^{cdef},$

$R^{ac}R^{b}{}_{c} - \frac{1}{2}g^{ab}R_{cd}R^{cd} - \frac{1}{2}R^{ab}R + \frac{1}{8}g^{ab}R^2 + R^{cd}R^{a}{}_{c}{}^{b}{}_{d} - \frac{1}{2}R^{acde}R^{b}{}_{cde} + \frac{1}{8}g^{ab}R_{cdef}R^{cdef},$

$R^{ac}R^{b}{}_{c} - \frac{1}{2}g^{ab}R_{cd}R^{cd} - \frac{1}{2}R^{ab}R + \frac{1}{8}g^{ab}R^2 + R^{cd}R^{a}{}_{c}{}^{b}{}_{d} - R^{acde}R^{b}{}_{dce} + \frac{1}{4}g^{ab}R_{cedf}R^{cdef}\}$

31

This constructed all dimensionally dependent identities that have two Riemann tensors (or contractions thereof) and free indices *a* and *b*. All of these four expression are zero. Even though there are four identities, only two of them are independent (not taking Bianchi identities into account). This can be verified with SolveTensors:

```
In := ddisols = SolveTensors[
         ddis == 0,
         UseSymmetries -> False, MetricOn -> None
      ]
```
32

Out = {{HoldPattern[$R^{\underline{\underline{acde}}}R^{\underline{\underline{b}}}{}_{\underline{\underline{cde}}}$] :→ Module[{f, h, i, g, j, k, l, m},

$2R^{af}R^{b}{}_{f} - g^{ab}R_{hi}R^{hi} - R^{ab}R + \frac{1}{4}g^{ab}R^2 + 2R^{fg}R^{a}{}_{f}{}^{b}{}_{g} + \frac{1}{4}g^{ab}R_{jklm}R^{jklm}$],

HoldPattern[$R^{\underline{\underline{acde}}}R^{\underline{\underline{b}}}{}_{\underline{\underline{dce}}}$] :→ Module[{f, h, i, g, j, k, l, m},

$R^{af}R^{b}{}_{f} - \frac{1}{2}g^{ab}R_{hi}R^{hi} - \frac{1}{2}R^{ab}R + \frac{1}{8}g^{ab}R^2 + R^{fg}R^{a}{}_{f}{}^{b}{}_{g} + \frac{1}{4}g^{ab}R_{jlkm}R^{jklm}$]}}

The two options are needed to prevent `SolveTensors` from making rules for every index combination on the left-hand-side related by symmetries (`UseSymmetries`) and by raising and lowering of the indices (`MetricOn`). Because SolveTensors returns a solution for two tensor structures in terms of others, only two of the four found DDIs are independent.[3]

The above output consists of rules that we can use to enforce the identities on the equations of motion:

```
In := eom /. ddisols // ToCanonical
```
Out = {0}

33

So, indeed, the equations of motion are zero.

[3] Again, this is true up to Bianchi identities. If we take those into account, there is only one truly independent DDI because $R_{acde}R^{bdce} = \frac{1}{2}R_{acde}R^{bcde}$. This identity is derived in Section 5.3.2.

Alternatively, we could have made an Ansatz with arbitrary coefficients from the identities,

In := `ddiAnsatz = CollectTensors @ MakeAnsatz[ddis]`

Out = $(C_1 + C_2 + C_3)R^{ac}R^{b}{}_{c} + \frac{1}{2}(-C_1 - C_2 - C_3)g^{ab}R_{cd}R^{cd} + \frac{1}{2}(-C_1 - C_2 - C_3)R^{ab}R$
$+\frac{1}{8}(C_1 + C_2 + C_3)g^{ab}R^2 + (C_1 + C_2 + C_3)R^{cd}R^{a}{}_{c}{}^{b}{}_{d} + (-\frac{1}{2}C_1 - C_2 + C_4)R^{acde}R^{b}{}_{cde}$
$+(C_2 - C_3 - 2C_4)R^{acde}R^{b}{}_{dce} + \frac{1}{8}(C_1 + 2C_2 - 2C_4)g^{ab}R_{cdef}R^{cdef}$
$+\frac{1}{4}(-C_2 + C_3 + 2C_4)g^{ab}R_{cedf}R^{cdef}$ 34

and tried to make this equal to the equations of motion by solving for the coefficients:

In := `SolveConstants[eom == ddiAnsatz]`

Out = $\{\{C_3 \to -4 - C_1 - C_2,\ C_4 \to 2 + \frac{1}{2}C_1 + C_2\}\}$ 35

So again, the equations of motion are equal to particular linear combination of the dimensionally dependent identities, and hence they are zero.

5. xTras functions

This section documents the most important functions in *xTras*. The list of functions below is not exhaustive, nor are the functions described in full detail (for example, most options are not described here). For a complete list of functions and all their options, please refer to the built-in documentation or the online documentation at www.xact.es/xtras/documentation.

Throughout this section, we assume we have a manifold `M`, a metric `metric`, a covariant derivative `CD` and associate curvature tensors (`RiemannCD`, `RicciCD`, etc.). These can be defined with the commands

In := `DefManifold[M, dim, IndexRange[a,m]]`

In := `DefMetric[-1, metric[-a,-b], CD, PrintAs->"g"]` 36

where `dim` is a predefined constant symbol.

5.1. Combinatorics

In this section we discuss some of the *xTras* functions that are of a combinatorial nature.

5.1.1. AllContractions

`AllContractions[`*expr*`]`
returns a sorted list of all possible full contractions of *expr* over its free indices.

`AllContractions[`*expr*`,` *frees*`]`
returns all possible contractions of *expr* that have *frees* as free indices.

`AllContractions[`*expr*`,` *frees*`,` *sym*`]`
returns all possible contractions of *expr* with the symmetry *sym* imposed on the free indices *frees*.

Details. A recurring problem in field theory is to make the most general Ansatz that contains a specific set of fields and derivatives. For constructing for example the most general gauge-invariant action for a particular set of fields (like we did for the free spin-2 field on a flat background in Section 4.1), one would need to know all possible vertices and all possible gauge transformations. While this problem is still tractable at lowest orders, it becomes complicated very fast at higher orders. In fact, the naive number of possible contractions of a tensorial expression that has n free indices is $(n-1)!!$, which is the number of independent products of $\frac{n}{2}$ metrics. The problem of finding all contractions when n is large is, if not error-prone, tedious at the very least. That is where the command `AllContractions` comes in.

The problem of finding all possible contractions of the input expression is equivalent to enumerating all double coset representatives of $K \setminus S_n/H$, where n is the number of indices of the input expression, K its symmetry group, and H the symmetry group of $\frac{n}{2}$ metrics. However, double coset enumeration is known the be an NP-hard problem in general [17], and to the author's knowledge no satisfactory algorithm has been found to date.

So instead of doing a proper double coset enumeration, `AllContractions` uses a brute-force-method to find all contractions. The algorithm it uses is as follows:

1. Take all single contractions of the input expression.
2. Canonicalize the single contractions, and throw away duplicates.
3. Take all second contractions of the canonicalized single contractions.
4. Canonicalize the second contractions, and throw away duplicates.
5. …

…and so on and so forth until all indices are contracted. This algorithm is reasonably fast if the input expression has a large degree of symmetry, but in general it is exponential in the number of indices to be contracted.

Examples. In its most basic form, `AllContractions` takes a single argument and computes all of its possible independent full contractions. Take for instance the Riemann tensor:

```
In := AllContractions[RiemannCD[-a, -b, -c, -d]]
Out = {R}
```
37

As we could have expected, its only possible full contraction is the Ricci scalar. If we take two Riemann tensors, things get a bit more interesting:

```
In := AllContractions[
        RiemannCD[-a, -b, -c, -d] RiemannCD[-e, -f, -g, -h]
      ]
```
38

Out = $\{R_{ab}R^{ab},\ R^2,\ R_{abcd}R^{abcd},\ R_{acbd}R^{abcd}\}$

The last two contractions are actually not independent, but are related via the Bianchi identity. The Bianchi identity is a multi-term symmetry, and `AllContractions` does not take these symmetries into account. Hence `AllContractions` does not necessarily return an irreducible basis of contractions, but it does always return a complete basis.

It is also possible to ask for contractions of expressions with derivatives:

```
In := AllContractions[ RicciCD[a, b] CD[c]@CD[d]@RicciCD[e, f] ]
```
39

Out = $\{R\nabla_a\nabla^a R,\ R\nabla_b\nabla_a R^{ab},\ R^{ab}\nabla_b\nabla_a R,\ R^{ab}\nabla_b\nabla_c R_a{}^c,\ R^{ab}\nabla_c\nabla_b R_a{}^c,\ R^{ab}\nabla_c\nabla^c R_{ab}\}$

Note that besides not taking Bianchi identities into account, `AllContractions` also does not automatically sort covariant derivatives.

`AllContractions` takes an optional second argument, which specifies what free indices the final contractions should have. This effectively adds an auxiliary tensor in the first argument with the specified indices, and varies the contractions afterwards with respect to this auxiliary tensor. Here's an example with two free indices:

```
In := AllContractions[
        RiemannCD[-a, -b, -c, -d] RiemannCD[-e, -f, -g, -h],
        {-a, -b}
      ]
```
40

Out = $\{R_a{}^c R_{bc},\ g_{ab}R_{cd}R^{cd},\ R_{ab}R,\ g_{ab}R^2,\ R^{cd}R_{acbd},\ R_a{}^{cde}R_{bcde},\ R_a{}^{cde}R_{bdce},\ g_{ab}R_{cdef}R^{cdef},\ g_{ab}R_{cedf}R^{cdef}\}$

We can also specify an optional third argument to `AllContractions`. This third argument specifies the symmetry of the indices in the second argument. For instance, we can try to see if there are any antisymmetric contractions in the above example:

```
In := AllContractions[
        RiemannCD[-a,-b,-c,-d] RiemannCD[-e,-f,-g,-h],
        {-a, -b},
        Antisymmetric[{-a,-b}]
      ]
Out = {}
```
41

As is obvious from the previous example, there are none.

5.1.2. *MakeTraceless*

`MakeTraceless[`*expr*`]`
returns the traceless version of *expr*.

Any tensor can be projected onto its irreducible traceless components. The way to do this by hand is to write down all possible traces of the tensor, make an Ansatz for a linear combination of them, and then demand that single traces of this Ansatz are zero. Needless to say, for tensors of large rank this task is perfectly suited for the computer.

The function `MakeTraceless` does exactly this: it takes its argument and makes it traceless. For the Ricci tensor, it gives the traceless Ricci tensor:

```
In := MakeTraceless[RicciCD[-a, -b]]
```

$$\text{Out} = R_{ab} - \frac{g_{ab}R}{d} \tag{42}$$

And if we enter the Riemann tensor, it returns the Weyl tensor:

```
In := MakeTraceless[RiemannCD[-a, -b, -c, -d]]
```

$$\text{Out} = R_{abcd} + \frac{2}{(d-2)(d-1)} R \underset{1234}{\mathrm{Sym}}(g_{ac}g_{bd}) - \frac{4}{-2+d}\underset{1234}{\mathrm{Sym}}(g_{bd}R_{ac}) \tag{43}$$

`MakeTraceless` uses the power of the *SymManipulator* package [12] to implicitly impose symmetry of the Riemann tensor without expanding all required terms. This is what the `Sym` objects in the above output do. We can remove them and expand all terms with the command `ExpandSym`, thereby recovering the usual expression for the Weyl tensor:

```
In := MakeTraceless[RiemannCD[-a,-b,-c,-d]] // ExpandSym // ToCanonical
```

$$\text{Out} = -\frac{g_{bd}R_{ac}}{-2+d} + \frac{g_{bc}R_{ad}}{-2+d} + \frac{g_{ad}R_{bc}}{-2+d} - \frac{g_{ac}R_{bd}}{-2+d} - \frac{g_{ad}g_{bc}R}{2-3d+d^2} + \frac{g_{ac}g_{bd}R}{2-3d+d^2} + R_{abcd} \tag{44}$$

We can convert this to the actual Weyl tensor with the *xTensor* command `RiemannToWeyl`:

```
In := RiemannToWeyl[%] // ToCanonical // Simplify
```

$$\text{Out} = W[\nabla]_{abcd} \tag{45}$$

`MakeTraceless` works on any expression without dummy indices. For example, here is the traceless version of a generic rank-3 tensor:

```
In := DefTensor[T[a,b,c], M]
In := MakeTraceless[T[a, b, c]]
```

$$\begin{aligned}\text{Out} = {} & T^{abc} - \frac{(1+d)g^{bc}T^{ad}{}_{d}}{-2+d+d^2} + \frac{g^{ac}T^{bd}{}_{d}}{-2+d+d^2} + \frac{g^{ab}T^{cd}{}_{d}}{-2+d+d^2} + \frac{g^{bc}T^{da}{}_{d}}{-2+d+d^2} \\ & - \frac{(1+d)g^{ac}T^{db}{}_{d}}{-2+d+d^2} + \frac{g^{ab}T^{dc}{}_{d}}{-2+d+d^2} + \frac{g^{bc}T^{d}{}_{d}{}^{a}}{-2+d+d^2} + \frac{g^{ac}T^{d}{}_{d}{}^{b}}{-2+d+d^2} - \frac{(1+d)g^{ab}T^{d}{}_{d}{}^{c}}{-2+d+d^2}\end{aligned} \tag{46}$$

We can extract the traceless projector on any rank-3 tensor by varying the above with respect to *T*:

```
In := VarD[T[d, e, f]] @ MakeTraceless[T[a, b, c]]
```

$$\begin{aligned}\text{Out} = {} & \delta^{a}{}_{d}\delta^{b}{}_{e}\delta^{c}{}_{f} - \frac{(1+d)\delta^{c}{}_{f}g^{ab}g_{de}}{-2+d+d^2} + \frac{\delta^{b}{}_{f}g^{ac}g_{de}}{-2+d+d^2} + \frac{\delta^{a}{}_{f}g^{bc}g_{de}}{-2+d+d^2} \\ & + \frac{\delta^{c}{}_{e}g^{ab}g_{df}}{-2+d+d^2} - \frac{(1+d)\delta^{b}{}_{e}g^{ac}g_{df}}{-2+d+d^2} + \frac{\delta^{a}{}_{e}g^{bc}g_{df}}{-2+d+d^2} + \frac{\delta^{c}{}_{d}g^{ab}g_{ef}}{-2+d+d^2} \\ & + \frac{\delta^{b}{}_{d}g^{ac}g_{ef}}{-2+d+d^2} - \frac{(1+d)\delta^{a}{}_{d}g^{bc}g_{ef}}{-2+d+d^2}\end{aligned} \tag{47}$$

This projector can then subsequently be used to make other rank-3 tensors traceless without having to call `MakeTraceless` again.

5.1.3. ConstructDDIs

`ConstructDDIs[`*expr*`]`
 constructs all scalar dimensional dependent identities that can be build out of *expr*.

`ConstructDDIs[`*expr*`, `*frees*`]`
 constructs all dimensional dependent identities that can be build out of *expr* and that have free indices *frees*.

`ConstructDDIs[`*expr*`, `*frees*`, `*sym*`]`
 constructs all dimensional dependent identities that can be build out of *expr* and that have the symmetry *sym* imposed on their free indices *frees*.

Details. Dimensional dependent identities (DDIs) are identities that only hold in specific dimensions. Typically, they can be derived from over-antisymmetrizations: that is, antisymmetrization over more indices than the number of dimensions. In d dimensions, one such identity is for example the generalized Kronecker delta with $2(d+1)$ indices:

$$\delta^{b_1\cdots b_{d+1}}_{a_1\cdots a_{d+1}} = (d+1)!\,\delta^{b_1}_{[a_1}\delta^{b_2}_{a_2}\cdots\delta^{b_d}_{a_d}\delta^{b_{d+1}}_{a_{d+1}]} = 0. \tag{48}$$

By contracting this identity with other tensors, it is possible to construct derived identities. For instance, in three dimensions we can contract it with a traceless $\{2,2\}$ tensor, such as the Weyl tensor, and find

$$\delta^e_{[a}\delta^f_b\delta^g_c\delta^h_{d]}W_{gh}{}^{ij} = \delta^e_{[a}\delta^f_b W_{cd]}{}^{ij} = 0. \tag{49}$$

Contracting over d and j gives

$$\delta^{[e}_{[a}W_{bc]}{}^{f]i} = 0, \tag{50}$$

and a further contraction over i and c gives the well-known fact that they Weyl tensor identically vanishes in three dimensions:

$$W_{ab}{}^{ef} = 0. \tag{51}$$

All DDIs that stem from over-antisymmetrizations can in fact be derived from the 'basic' identity (48) because it is always possible to pull out deltas on the over-antisymmetrized indices. Over-antisymmetrization over more than $d+1$ indices will give not give independent DDIs, because they can be written as linear combinations of antisymmetrizations over $d+1$ indices.

A systematic way of enumerating all DDIs is to consider all possible contractions of the fundamental identity (48) with the relevant tensors. This is exactly what `ConstructDDIs` does: it computes via AllContractions all contractions between the input expression and the basic identity (48) in the relevant dimension. In performing these contractions, two observations make life computationally easier: the independent index configurations of the basic identity are given by its standard Young tableaux, and the basic identity is completely traceless.

The latter is important for the following reason. While we can still write down meaningful derived identities with the uncontracted basic identity, this is not possible with any of its contractions — attempting to do so results in the trivial statement $0 = 0$. The difference between the vanishing of the uncontracted and the contracted basic identity is that the former is identically zero only when explicitly writing the indices out as $a, b, \ldots \in \{0, \ldots, d-1\}$, whereas the latter is identically zero without doing so.

To see why the basic identity is traceless, consider for example the basic identity in one dimension:

$$g_{a[b}g_{c]d} = 0. \tag{52}$$

Writing out the antisymmetrization and contracting a pair of indices gives

$$\tfrac{1}{2}g^{ab}\left(g_{ab}g_{cd} - g_{ac}g_{bd}\right) = \frac{d-1}{2}g_{cd} = 0, \tag{53}$$

where d is the number of dimensions. Doing the same exercise for the basic identity in two dimensions gives

$$g^{ab}g_{[\underline{a}b}\,g_{\underline{c}d}\,g_{\underline{e}]f} = \frac{d-2}{3}g_{[\underline{c}d}\,g_{\underline{e}]f} = 0, \tag{54}$$

while three dimensions gives

$$g^{ab}g_{[\underline{a}b}\,g_{\underline{c}d}\,g_{\underline{e}f}\,g_{\underline{g}]h} = \frac{d-3}{4}g_{[\underline{c}d}\,g_{\underline{e}f}\,g_{\underline{g}]h} = 0. \tag{55}$$

The same holds true for other contractions. Thus the fact that the basic identity is traceless is a dimensionally dependent statement.

The tracelessness of the basic identity allows us to only consider contractions of the form

$$\delta^{a_1\cdots a_{2(d+1)}}\,\langle x\rangle_{a_1\cdots a_{2(d+1)}}, \tag{56}$$

where $\delta^{a_1\cdots a_{2(d+1)}}$ is the basic identity (48), and by $\langle x\rangle_{a_1\cdots a_{2(d+1)}}$ we mean all contractions of x with $2(d+1)$ free indices. Taking all possible combinations of these contractions with the standard Young tableaux of the basic identity then yields all (scalar) DDIs.

Examples. In two dimensions, the Einstein tensor vanishes. This can be reproduced by asking for all DDIs with two free indices constructed out of the Riemann tensor:

```
In := dim = 2
In := ConstructDDIs[RiemannCD[a, b, c, d], {a, b}]                57
```

Out = $\{R^{ab} - \frac{1}{2}g^{ab}R\}$

Note that `ConstructDDIs` returns a list of expressions that are zero, and not equations.

In three dimensions, the Weyl tensor is zero. This time, we need four free indices that have the symmetry of the Riemann tensor:

```
In := dim = 3
In := ConstructDDIs[
        RiemannCD[a, b, c, d],
        {a, b, c, d},
        RiemannSymmetric[{a, b, c, d}]
      ]
```

Out = $\{g^{bd}R^{ac} - g^{bc}R^{ad} - g^{ad}R^{bc} + g^{ac}R^{bd} + \frac{1}{2}g^{ad}g^{bc}R - \frac{1}{2}g^{ac}g^{bd}R - R^{abcd},$
$g^{bd}R^{ac} - g^{bc}R^{ad} - g^{ad}R^{bc} + g^{ac}R^{bd} + \frac{1}{2}g^{ad}g^{bc}R - \frac{1}{2}g^{ac}g^{bd}R - 2R^{abcd} + R^{acbd} - R^{adbc},$
$g^{bd}R^{ac} - g^{bc}R^{ad} - g^{ad}R^{bc} + g^{ac}R^{bd} + \frac{1}{2}g^{ad}g^{bc}R - \frac{1}{2}g^{ac}g^{bd}R - R^{acbd} + R^{adbc},$
$R^{abcd} - R^{acbd} + R^{adbc}\}$ 58

Converting the above to Weyl tensors, we find:

```
In := % // RiemannToWeyl // CollectTensors
```

Out = $\{-W[\triangledown]^{abcd},\ -2W[\triangledown]^{abcd} + W[\triangledown]^{acbd} - W[\triangledown]^{adbc},\ -W[\triangledown]^{acbd} + W[\triangledown]^{adbc},$
$W[\triangledown]^{abcd} - W[\triangledown]^{acbd} + W[\triangledown]^{adbc}\}$ 59

As is obvious from this example, `ConstructDDIs`, like AllContractions, does not take multi-term symmetries like the Bianchi identity into account.

5.1.4. IndexConfigurations

`IndexConfigurations[`*expr*`]`
gives a list of all independent index configurations of *expr*.

Details. The command `IndexConfigurations` gives all possible independent permutations of the free indices of the input expression. A permutation of the free indices (or index configuration) is independent when it cannot be related to another index configuration by canonicalizing. The heavy lifting in `IndexConfigurations` is actually done by the *SymManipulator* package [12], which can compute the right transversal of H in S_n, where H is the symmetry group of the input expression, and n the number of free indices. A right transversal is the set of representatives of the right cosets $H\backslash S_n$, which in turn is in one-to-one correspondence to the set of independent index configurations.

Examples. Here's one simple example of how to use `IndexConfigurations`:

```
In := IndexConfigurations[metric[a, b]]
```

Out = $\{g^{ab}\}$ 60

Because the metric is symmetric, there is only one index configuration. For two metrics we get:

```
In := IndexConfigurations[metric[a, b] metric[c, d]]
```

Out = $\{g^{ad}g^{bc},\ g^{ac}g^{bd},\ g^{ab}g^{cd}\}$ 61

And for three metrics:

```
In := IndexConfigurations[metric[a, b] metric[c, d] metric[e, f]]
```

Out = $\{g^{af}g^{be}g^{cd},\ g^{ae}g^{bf}g^{cd},\ g^{af}g^{bd}g^{ce},\ g^{ad}g^{bf}g^{ce},\ g^{ae}g^{bd}g^{cf},$
$g^{ad}g^{be}g^{cf},\ g^{af}g^{bc}g^{de},\ g^{ac}g^{bf}g^{de},\ g^{ab}g^{cf}g^{de},\ g^{ae}g^{bc}g^{df},$
$g^{ac}g^{be}g^{df},\ g^{ab}g^{ce}g^{df},\ g^{ad}g^{bc}g^{ef},\ g^{ac}g^{bd}g^{ef},\ g^{ab}g^{cd}g^{ef}\}$ 62

Lastly, for the Riemann tensor we obtain:

```
In := IndexConfigurations[RiemannCD[-a, -b, -c, -d]]
```

Out = $\{R_{abcd},\ R_{acbd},\ R_{adbc}\}$ 63

Note that `IndexConfigurations` does not take multi-term symmetries like the Bianchi identity into account, and hence it does not see that the last term can actually be written in terms of the first two.

5.1.5. MakeAnsatz

`MakeAnsatz[{`$e_1, e_2, \cdots$`}]`
returns $C_1 e_1 + C_2 e_2 + \ldots$, where the C_i's are newly defined constant symbols.

`MakeAnsatz` is a convenience function that, out of a list of terms, constructs an Ansatz with constant Symbols. Here's an example of how it works:

```
In := MakeAnsatz[{metric[-a, -b], RicciCD[-a, -b]}]
```
Out = $C_1 g_{ab} + C_2 R_{ab}$ 64

Even though the constant symbols print as C_i, their Mathematica symbol name is `Ci`:

```
In := {C1, C2}
```
Out = $\{C_1, C_2\}$ 65

In combination with other functions such as IndexConfigurations or AllContractions, `MakeAnsatz` becomes very handy:

```
In := MakeAnsatz @ IndexConfigurations[metric[a, b] metric[c, d]]
```
Out = $C_1 g^{ad} g^{bc} + C_2 g^{ac} g^{bd} + C_3 g^{ab} g^{cd}$ 66

```
In := MakeAnsatz @ AllContractions[
         RiemannCD[a, b, c, d] RiemannCD[e, f, g, h]
      ]
```
Out = $C_1 R_{ab} R^{ab} + C_2 R^2 + C_3 R_{abcd} R^{abcd} + C_4 R_{acbd} R^{abcd}$ 67

5.2. Tensor algebra

This section describes the functions in *xTras* that can be used for doing basic algebra with tensors. There are two functions for rewriting expressions (CollectTensors and CollectConstants), and two functions for solving equations (SolveTensors and SolveConstants).

5.2.1. CollectTensors

`CollectTensors[`*expr*`]`
collects all tensorial terms in *expr*.

`CollectTensors` works like the Mathematica function `Collect`, with the difference that you do not have to specify a second argument: it collects all tensorial terms it can find in the input expression. A 'tensorial term' is a single tensor, or a product of tensors that cannot be expanded into a sum.

For example, assuming the scalars `X[]`, `Y[]`, and `Z[]` are defined, we can make the following expression:

```
In := expr = MakeAnsatz[
         {X[], X[], Y[], Y[], Z[], Z[], X[] Y[], X[] Y[]}
      ]
```
Out = $C_1 X + C_2 X + C_3 Y + C_4 Y + C_7 XY + C_8 XY + C_5 Z + C_6 Z$ 68

By calling `CollectTensors`, the tensors in this expression will be collected together:

```
In := CollectTensors[expr]
```
Out = $(C_1 + C_2) X + (C_3 + C_4) Y + (C_7 + C_8) XY + (C_5 + C_6) Z$ 69

`CollectTensors` also handles non-scalar tensors, which by default will be canonicalized before being collected:

```
In := CollectTensors[
         C1 RicciCD[-b, -a] + C2 metric[-a, -c] RicciCD[c, -b]
      ]
```
Out = $(C_1 + C_2) R_{ab}$ 70

5.2.2. *CollectConstants*

`CollectConstants[`*expr*`]`
collects all constant symbols in *expr*.

`CollectConstants` is the sibling of CollectTensors. Instead of collecting all tensorial terms in the input expression, it collects all constant symbols it can find. For example:

```
In := CollectConstants[
         C1 X[] + C1 Y[] + C2 Z[] + C2 X[] + C3 Z[] + C3 X[] Y[]
      ]                                                              71
```
Out = $C_1(X+Y)+C_2(X+Z)+C_3(XY+Z)$

5.2.3. *SolveConstants*

`SolveConstants[`*expr*`]`
attempts to solve the system *expr* of tensorial equations for all constant symbols appearing in *expr*.

The function `SolveConstants` solves equations with respect to constant symbols. Not only does it do that, it also makes sure no tensors appear on the right-hand-side of the solutions. To achieve this, it uses the following three-step procedure:

1. Use `CollectTensors` on the equation to group tensorial terms.
2. Read of equations for the prefactors from each of the tensorial terms.
3. Solve the prefactor equations simultaneously with built-in Mathematica function `Solve`.

To illustrate this procedure, take for example the same expression we had in Section 5.2.1, namely

```
In := expr = MakeAnsatz[
         {X[], X[], Y[], Y[], Z[], Z[], X[] Y[], X[] Y[]}
      ]                                                              72
```
Out = $C_1X+C_2X+C_3Y+C_4Y+C_7XY+C_8XY+C_5Z+C_6Z$

The first step towards solving the equation `expr == 0` for the constant symbols C_i is to collect the tensorial terms:

```
In := CollectTensors[expr]                                           73
```
Out = $(C_1+C_2)X+(C_3+C_4)Y+(C_7+C_8)XY+(C_5+C_6)Z$

The second step is to read off equations for the constant symbols from each tensorial term. The *xTras* function `ToConstantSymbolEquations` does exactly this:

```
In := ToConstantSymbolEquations[% == 0]                              74
```
Out = $C_1+C_2 == 0$ && $C_3+C_4 == 0$ && $C_5+C_6 == 0$ && $C_7+C_8 == 0$

The result is then fed into `Solve`:

```
In := Solve[%, {C2, C4, C6, C8}]                                     75
```
Out = $\{\{C_2\to -C_1,\ C_4\to -C_3,\ C_6\to -C_5,\ C_8\to -C_7\}\}$

Indeed, this is the same answer we would have gotten if we had directly asked `SolveConstants`:

```
In := SolveConstants[expr == 0]                                      76
```
Out = $\{\{C_2\to -C_1,\ C_4\to -C_3,\ C_6\to -C_5,\ C_8\to -C_7\}\}$

5.2.4. *SolveTensors*

`SolveTensors[`*expr*`]`
attempts to solve the system *expr* of tensorial equations for all tensors in *expr*.

`SolveTensors[`*expr*`, `*tens*`]`
attempts to solve the system *expr* of tensorial equations for the tensors *tens*.

Solving equations for tensors in an automated fashion is a tricky proposition. Not only does one have to deal with dummy indices and different forms of tensors, but also with the fact the equations may be solved only after taking one or more contractions. `SolveTensors` does not address these issues; instead, it rather solves tensorial equations for any (product of) tensor(s) that is not contracted with another tensor. This does not always return the most general space of solutions, but a subset of it.

For example, it solves the Einstein equation as

```
In := SolveTensors[
         RicciCD[-a, -b] - 1/2 metric[-a, -b] RicciScalarCD[] == 0
      ]                                                                 77
```

Out = $\{\{\text{HoldPattern}[R^{\underline{\underline{ab}}}] :\!> \text{Module}[\{\}, \frac{1}{2}g^{ba}R]\}\}$

The double line underneath the indices on the left-hand-side ensures that all Ricci tensors get replaced when using this rule, regardless whether their indices are up or down:

```
In := RicciCD[c, -d] /. %                                               78
```

Out = $\{\frac{1}{2}\delta_d{}^c R\}$

In some simple cases, `SolveTensors` does return the general solution,

```
In := SolveTensors[Z[] X[] == Z[] Y[]]
Out= {{HoldPattern[Y] :> Module[{}, X]},                                79
      {HoldPattern[Z] :> Module[{}, 0]}}
```

but, as said, in general it does not. Hence `SolveTensors` should more be used as a way to easily obtain proper *xAct* tensor replacement rules than as a method to solve generic tensorial equations.

It is worth mentioning that the second argument of `SolveTensors`, which specifies what tensors to solve for, also takes patterns:

```
In := SolveTensors[
         RicciCD[-a, -b] - 1/2 metric[-a, -b] RicciScalarCD[] == 0,
         metric[__]
      ]                                                                 80
```

Out = $\{\{\text{HoldPattern}[g^{\underline{\underline{ab}}}] :\!> \text{Module}[\{\}, \frac{2R^{ab}}{R}]\}\}$

Because the pattern `metric[__]` matches the explicit form `metric[-a, -b]`, this solved for the metric. For higher rank tensors using patterns is particularly convenients, as this avoids having to type all indices.

5.3. Young tableaux

Conspicuously absent in *xAct* are functions that deal with Young tableaux and multi-term symmetries. *xTras* provides a few functions in an attempt to partly fill this void, but it is by no means a complete treatment of the subject.

5.3.1. YoungProject

`YoungProject[`*expr*`, `*tab*`]`
projects the tensorial expression *expr* onto the Young tableau *tab*.

Details. If you try to antisymmetrize the Riemann tensor over three indices in *xAct*, you will find that the result is non-zero:

```
In := ToCanonical @ Antisymmetrize[RiemannCD[-a,-b,-c,-d], {-a,-b,-c}]  81
```

Out = $\frac{1}{3}R_{abcd} - \frac{1}{3}R_{acbd} + \frac{1}{3}R_{adbc}$

This is because `ToCanonical` does not take multi-term symmetries, like the Bianchi identity $R_{[abc]d} = 0$, into account. However, these symmetries can be made explicit by projecting tensors onto their respective Young tableaux [18]. The projection can be done with so-called Young projectors [19], which are sequential row-by-row symmetrizations and column-by-column antisymmetrizations of the Young tableau. To be precise, if we have a Young diagram λ (i.e. a partition of the integer n) and one of its Young tableaux λ_a, then the Young projector reads

$$P_A^{\lambda_a} = \frac{f^\lambda}{n!} \prod_{k\in\text{col}(\lambda_a)} A^k \prod_{l\in\text{row}(\lambda_a)} S^l \tag{82}$$

where f^λ is the dimension of the Young diagram, and S^n (A^n) the (anti-)symmetrization of the n^{th} row (column). Here, both S and A are without any weight, i.e. $S(\{x, y\}) = \{x, y\} + \{y, x\}$ and not $\frac{1}{2}(\{x, y\} + \{y, x\})$.

The above Young projector is manifestly antisymmetric, because the columns are antisymmetrized after the rows are symmetrized. Changing this order gives the manifestly symmetric Young projector P_S:

$$P_S^{\lambda_a} = \frac{f^\lambda}{n!} \prod_{l\in\text{row}(\lambda_a)} S^l \prod_{k\in\text{col}(\lambda_a)} A^k \tag{83}$$

which is also a perfectly fine projector. By default, `YoungProject` uses the manifestly antisymmetric projector P_A, but by setting the option `ManifestSymmetry` to `Symmetric` it is possible to use the manifestly symmetric projector P_S.

Examples. Projecting a tensor S^{ab} onto the Young tableau $\boxed{a}\boxed{b}$ can be done as follows:

```
In := YoungProject[S[a, b], {{a, b}}]
```

Out = $\frac{1}{2}S^{ab} + \frac{1}{2}S^{ba}$ 84

And projecting it onto the tableau $\begin{array}{|c|}\hline a\\\hline b\\\hline\end{array}$ gives:

```
In := YoungProject[S[a, b], {{a}, {b}}]
```

Out = $\frac{1}{2}S^{ab} - \frac{1}{2}S^{ba}$ 85

Projecting the Riemann tensor onto the tableau $\begin{array}{|c|c|}\hline a & c\\\hline b & d\\\hline\end{array}$ goes as follows:

```
In := YoungProject[RiemannCD[-a, -b, -c, -d], {{-a, -c}, {-b, -d}}]
```

Out = $\frac{2}{3}R_{abcd} + \frac{1}{3}R_{acbd} - \frac{1}{3}R_{adbc}$ 86

And indeed, the Bianchi identity is manifest after projection:

```
In := ToCanonical @ Antisymmetrize[%, {-a, -b, -c}]
Out = 0
```
87

By default, `YoungProject` uses a manifestly antisymmetric projection. It projects for example a rank-3 tensor T^{abc} onto the Young tableau $\begin{array}{|c|c|}\hline a & b\\\hline c\\\cline{1-1}\end{array}$ as

```
In := YoungProject[T[a, b, c], {{a, b}, {c}}]
```

Out = $\frac{1}{3}T^{abc} + \frac{1}{3}T^{bac} - \frac{1}{3}T^{bca} - \frac{1}{3}T^{cba}$ 88

which is indeed antisymmetric in a and c. We can switch to a manifestly symmetric projection with the option `ManifestSymmetry`:

```
In := YoungProject[
        T[a, b, c],
        {{a, b}, {c}},
        ManifestSymmetry -> Symmetric
      ]
```

Out = $\frac{1}{3}T^{abc} + \frac{1}{3}T^{bac} - \frac{1}{3}T^{cab} - \frac{1}{3}T^{cba}$ 89

The result is now no longer antisymmetric in a and c, but symmetric in a and b.

5.3.2. RiemannYoungProject

`RiemannYoungProject[`*expr*`]`
projects all Riemann tensors and their first derivatives in *expr* onto their Young tableaux.

The function `RiemannYoungProject` automatizes the projection of Riemann tensors onto their Young tableaux; it replaces every occurrence of a Riemann tensor or a first derivative of it with their Young projected versions. That is, it does the replacements

$$R_{abcd} \to P_A^{\begin{ytableau} a & c \\ b & d \end{ytableau}}(R_{abcd})\,, \tag{90a}$$

$$\nabla_e R_{abcd} \to P_A^{\begin{ytableau} a & c & e \\ b & d \end{ytableau}}(\nabla_e R_{abcd})\,. \tag{90b}$$

For example, a single Riemann tensor is replaced as follows:

```
In := RiemannYoungProject @ RiemannCD[-a, -b, -c, -d]
```

Out = $\frac{2}{3}R_{abcd} + \frac{1}{3}R_{acbd} - \frac{1}{3}R_{adbc}$ 91

A first derivative of the Riemann tensor gets replaced as:

```
In := RiemannYoungProject[CD[-e] @ RiemannCD[-a, -b, -c, -d]]
```

Out = $\frac{1}{12}\nabla_a R_{bcde} - \frac{1}{12}\nabla_a R_{bdce} - \frac{1}{6}\nabla_a R_{becd} - \frac{1}{12}\nabla_b R_{acde} + \frac{1}{12}\nabla_b R_{adce} + \frac{1}{6}\nabla_b R_{aecd}$
$-\frac{1}{6}\nabla_c R_{abde} - \frac{1}{12}\nabla_c R_{adbe} + \frac{1}{12}\nabla_c R_{aebd} + \frac{1}{6}\nabla_d R_{abce} + \frac{1}{12}\nabla_d R_{acbe} - \frac{1}{12}\nabla_d R_{aebc}$
$+\frac{1}{3}\nabla_e R_{abcd} + \frac{1}{6}\nabla_e R_{acbd} - \frac{1}{6}\nabla_e R_{adbc}$ 92

This enables us to easily prove e.g. the second Bianchi identity $\nabla_{[a}R_{bc]de} = 0$:

```
In := ToCanonical @ RiemannYoungProject @ Antisymmetrize[
        CD[-a] @ RiemannCD[-b, -c, -d, -e],
        {-a, -b, -c}
      ]
Out = 0
```
93

Another nice example is the identity $R_{acde}R^{bdce} = \frac{1}{2}R_{acde}R^{bcde}$, which can be proven as follows:

```
In := ToCanonical @ RiemannYoungProject[
        RiemannCD[-a,-c,-d,-e](RiemannCD[b,d,c,e]-1/2RiemannCD[b,c,d,e])
      ]
Out = 0
```
94

5.3.3. *TableauSymmetric*

`TableauSymmetric[`*tab*`]`
gives the symmetry of the tableau *tab*.

`TableauSymmetric` generalizes the *xAct* functions `Symmetric`, `Antisymmetric`, and `RiemannSymmetric` to arbitrary Young tableaux. This comes in particularly handy when defining tensors that have more complicated symmetry structures than just complete (anti-)symmetry. Say, for instance, we have a tensor T^{abcdef} that lives in the Young diagram $\ydiagram{3,2,1}$. If we define it without any symmetry,

```
In := DefTensor[T[a,b,c,d,e,f], M]
```
95

and subsequently project it onto its Young tableau, we get no less than 144 terms:

```
In := Length @ YoungProject[
        T[a, b, c, d, e, f],
        {{a, b, c}, {d, e}, {f}}
      ]
Out = 144
```
96

However, if we had instead defined it with the appropriate symmetry,

```
In := DefTensor[
        T[a,b,c,d,e,f], M,
        TableauSymmetric[{{a,b,c}, {d,e}, {f}}
      ]
```
97

we would have gotten just 57 terms:

```
In := Length @ YoungProject[
         T[a, b, c, d, e, f],
         {{a, b, c}, {d, e}, {f}}
      ]                                              98
Out = 57
```

This is because the tensor T^{abcdef} now has all the mono-term symmetries that come from its Young diagram. For example,

```
In := ToCanonical[T[f, e, c, a, b, d]]
                                                     99
Out = -T^abcdef
```

It are these mono-term symmetries that reduce the number of terms in the Young projection.

5.4. Miscellaneous

Lastly, this section describes some *xTras* functions that do not fall in any of the other categories.

5.4.1. VarL

`VarD[g[-a,-b], cd][S]`

returns $\frac{\delta S}{\delta g_{ab}}$ while integrating by parts with respect to the covariant derivative *cd*.

`VarL[g[-a,-b], cd][L]`

returns $\frac{1}{\sqrt{|g|}}\frac{\delta\sqrt{|g|}L}{\delta g_{ab}}$ while integrating by parts with respect to the covariant derivative *cd*.

Details. Because of the non-linear metric dependence of curvature tensors, computing their equations of motion with respect to the metric can be a rather involved affair. While the variation of the Einstein–Hilbert term is relatively easy, things like

$$\frac{\delta}{\delta g_{ab}}\left(R_{cd}{}^{gh}R^{cdef}\nabla_f\nabla_l R_h{}^l{}_{ik}\nabla_j\nabla_g R_e{}^{ijk}\right) \tag{100}$$

can be quite cumbersome. By using the power of the *xPert* package [7], *xTras* can compute variations like the above with relative ease. It does this by first computing the total variation, and then integrating by parts. Schematically, this reads

$$\delta F = f_1\delta g + f_2\nabla\delta g + f_3\nabla\nabla\delta g + \cdots = \frac{\delta F}{\delta g}\delta g + \text{total derivative} \tag{101}$$

where F and f_i are a functionals that depend on the metric g, and $\frac{\delta F}{\delta g}$ is the quantity we are after.

Computing the total variation is the first step towards reading off $\frac{\delta F}{\delta g}$, and is carried out by the *xPert* commands `Perturbation` and `ExpandPerturbation`:

```
In := ExpandPerturbation @ Perturbation[RicciScalarCD[]]
         // ContractMetric // ToCanonical
                                                     102
```
Out = $-\triangle g^{1ab}R_{ab} + \nabla_b\nabla_a\triangle g^{1ab} - \nabla_b\nabla^b\triangle g^{1a}{}_a$

Here $\triangle g^1_{ab}$ is the same as δg_{ab} above, namely the perturbation of the metric. The second step, integrating by parts and peeling off δg_{ab}, is done with the *xTensor* command `VarD`:

In := `VarD[`$\triangle g^1_{ab}$`, CD][%]` 103

Out = $-\delta_1{}^1 g^{ac}g^{bd}R_{cd}$

The spurious $\delta_1{}^1$ comes from the way *xAct* handles the variation $\frac{\delta\triangle g^1_{ab}}{\delta\triangle g^1_{cd}}$ and is equal to one, even though it is not automatically simplified.

xTras overwrites the `VarD` command such that the above two-step procedure is carried out whenever the variation is with respect to a metric. When the variation is with respect to another tensor, *xTensor*'s `VarD` is used.

Examples. The variation of the Ricci scalar with respect to the metric can be computed with the following command:

```
In := VarD[metric[-a, -b], CD][RicciScalarCD[]]
```

$$\text{Out} = -g^{ac}g^{bd}R_{cd} \qquad 104$$

Using `VarL` instead of `VarD` automatically takes care of overall factors of $\sqrt{|g|}$:

```
In := VarL[metric[-a, -b], CD][RicciScalarCD[]]
```

$$\text{Out} = -g^{ac}g^{bd}R_{cd} + \frac{1}{2}g^{ab}R \qquad 105$$

Note that `VarD` and `VarL` do not contract metrics and canonicalize on their own. If we want, we have to do this ourselves afterwards. Varying the Einstein–Hilbert term coupled to a scalar field ϕ with respect to the metric gives:

```
In := VarL[metric[-a, -b], CD][phi[] RicciScalarCD[]]
        // ContractMetric // ToCanonical
```

$$\text{Out} = -\phi R^{ab} + \frac{1}{2}g^{ab}\phi R + \frac{1}{2}\nabla^a\nabla^b\phi + \frac{1}{2}\nabla^b\nabla^a\phi - g^{ab}\nabla_c\nabla^c\phi \qquad 106$$

Higher powers of R can also be varied easily:

```
In := VarL[metric[-a, -b], CD][RicciScalarCD[]^2]
        // ContractMetric // ToCanonical
```

$$\text{Out} = -2R^{ab}R + \frac{1}{2}g^{ab}R^2 + \nabla^a\nabla^bR + \nabla^b\nabla^aR - 2g^{ab}\nabla_c\nabla^cR \qquad 107$$

And higher still:

```
In := VarL[metric[-a, -b], CD][RicciScalarCD[]^4]
        // ContractMetric // ToCanonical
```

$$\text{Out} = -4R^{ab}R^3 + \frac{1}{2}g^{ab}R^4 + 6R^2\nabla^a\nabla^bR + 24R\nabla^aR\nabla^bR + 6R^2\nabla^b\nabla^aR - 12g^{ab}R^2\nabla_c\nabla^cR$$
$$-24g^{ab}R\nabla_cR\nabla^cR \qquad 108$$

5.4.2. *FullSimplification*

`FullSimplification[][`*expr*`]`

tries to simplify *expr* as much as possible, taking Bianchi identities into account and sorting covariant derivatives.

When dealing with curvature tensors, it is often desirable to use the Bianchi identities to rewrite expression in the simplest form possible. `ToCanonical` cannot be used for this, since it only simplifies mono-term symmetries, and Bianchi identities are multi-term symmetries. The Bianchi identities are however implemented in the simplification methods of the *Invar* package [9,10]. But unfortunately, *Invar* can only simplify scalar monomials of Riemann tensors.

The function `FullSimplification` extends the capabilities of *Invar* slightly by also simplifying the contracted second Bianchi identities in any expression, not just scalar monomials. When given an input expression, `FullSimplification` does the following:

1. Simplify scalar monomials with the help of the *Invar* package.
2. Apply the contracted second Bianchi identities $\nabla_aR_{bcd}{}^a = \nabla_cR_{bd} - \nabla_bR_{cd}$ and $\nabla_aR_b{}^a = \frac{1}{2}\nabla_bR$.
3. Sort covariant derivatives.

For example, when given the expression $\nabla^a\nabla_bR_{ca}$, `FullSimplification` commutes covariant derivatives to divergences such that it can use the contracted Bianchi identities, and then afterwards sorts covariant derivatives:

```
In := FullSimplification[][CD[a] @ CD[-b] @ RicciCD[-c, -a]]
```

$$\text{Out} = R_b{}^aR_{ca} - R^{ad}R_{bacd} + \frac{1}{2}\nabla_c\nabla_bR \qquad 109$$

Note that covariant derivatives are sorted with the *xAct* command `SortCovDs`, which sorts them in alphabetical order in postfix notation. Thus $\nabla_c\nabla_bR = R_{;b;c}$ is sorted.

As said, `FullSimplification` also simplifies scalar monomials by using all Bianchi identities, not just the contracted Bianchi identities:

```
In := FullSimplification[][RiemannCD[a,b,c,d] RiemannCD[-a,-c,-b,-d]]
```

$$\text{Out} = \frac{1}{2}R_{abcd}R^{abcd} \qquad 110$$

This is a contraction of the identity we found in Section 5.3.2.

5.4.3. EulerDensity

`EulerDensity[cd]`

gives the Euler density associated to the covariant derivative *cd*.

`EulerDensity[cd, dim]`

gives the Euler density associated to the covariant derivative *cd* in the dimension *dim* if the underlying manifold has a generic dimension.

Details. The Euler density E_{2n} in dimension $d = 2n$ is given by

$$E_{2n} = \frac{1}{2^n} R_{i_1 i_2 j_1 j_2} \cdots R_{i_{n-1} i_n j_{n-1} j_n} \epsilon^{i_1 \cdots i_n} \epsilon^{j_1 \cdots j_n} \tag{111}$$

where ϵ is the Levi-Civita tensor, not the Levi-Civita symbol. Note that this technically is not a density because it has zero weight. In order to obtain a density, we would need to multiply it with $\sqrt{|g|}$.

In order to prevent dummy index collisions, the results of `EulerDensity` are wrapped in a special head `Scalar`, which is indicated by a bracket. The `Scalar` heads can be removed with the *xTensor* command `NoScalar`.

Examples. Because we have a manifold with generic dimension, we need to specify the second argument of `EulerDensity`. For two dimensions, the Euler density reads:

In := `EulerDensity[CD, 2]`

Out = $-R$ 112

And for four dimensions it is:

In := `EulerDensity[CD, 4]`

Out = $-R^2 + 4\left(R_{ab}R^{ab}\right) - \left(R_{abcd}R^{abcd}\right)$ 113

In six dimensions the Euler density becomes:

In := `EulerDensity[CD, 6]`

Out = $-R^3 + 12R\left(R_{ab}R^{ab}\right) - 16\left(R_{a}{}^{c}R^{ab}R_{bc}\right) - 24\left(R^{ab}R^{cd}R_{acbd}\right) - 3R\left(R_{abcd}R^{abcd}\right)$
$+24\left(R^{ab}R_{a}{}^{cde}R_{bcde}\right) + 8\left(R_{a}{}^{e}{}_{c}{}^{f}R^{abcd}R_{bfde}\right) - 2\left(R_{ab}{}^{ef}R^{abcd}R_{cdef}\right)$ 114

And lastly, in eight dimensions, it is:

In := `EulerDensity[CD, 8]`

Out = $-R^4 + 24R^2\left(R_{ab}R^{ab}\right) - 64R\left(R_{a}{}^{c}R^{ab}R_{bc}\right) + 96\left(R_{a}{}^{c}R^{ab}R_{b}{}^{d}R_{cd}\right) - 48\left(R_{ab}R^{ab}\right)\left(R_{cd}R^{cd}\right)$
$-96R\left(R^{ab}R^{cd}R_{acbd}\right) - 6R^2\left(R_{abcd}R^{abcd}\right) + 96R\left(R^{ab}R_{a}{}^{cde}R_{bcde}\right) + 384\left(R_{a}{}^{c}R^{ab}R^{de}R_{bdce}\right)$
$-96\left(R^{ab}R^{cd}R_{ac}{}^{ef}R_{bdef}\right) - 192\left(R^{ab}R^{cd}R_{a}{}^{e}{}_{c}{}^{f}R_{bedf}\right) + 32R\left(R_{a}{}^{e}{}_{c}{}^{f}R^{abcd}R_{bfde}\right)$
$-8R\left(R_{ab}{}^{ef}R^{abcd}R_{cdef}\right) - 192\left(R_{a}{}^{c}R^{ab}R_{b}{}^{def}R_{cdef}\right) + 192\left(R^{ab}R^{cd}R_{a}{}^{e}{}_{b}{}^{f}R_{cedf}\right)$ 115
$-384\left(R^{ab}R_{a}{}^{cde}R_{b}{}^{f}{}_{d}{}^{g}R_{cgef}\right) + 24\left(R_{ab}R^{ab}\right)\left(R_{cdef}R^{cdef}\right) + 96\left(R^{ab}R_{a}{}^{cde}R_{bc}{}^{fg}R_{defg}\right)$
$-192\left(R^{ab}R_{a}{}^{c}{}_{b}{}^{d}R_{c}{}^{efg}R_{defg}\right) + 96\left(R_{a}{}^{e}{}_{c}{}^{f}R^{abcd}R_{b}{}^{g}{}_{e}{}^{h}R_{dgfh}\right) + 96\left(R_{ab}{}^{ef}R^{abcd}R_{c}{}^{g}{}_{e}{}^{h}R_{dhfg}\right)$
$-6\left(R_{ab}{}^{ef}R^{abcd}R_{cd}{}^{gh}R_{efgh}\right) + 48\left(R_{abc}{}^{e}R^{abcd}R_{d}{}^{fgh}R_{efgh}\right) - 48\left(R_{a}{}^{e}{}_{c}{}^{f}R^{abcd}R_{b}{}^{g}{}_{d}{}^{h}R_{egfh}\right)$
$-3\left(R_{abcd}R^{abcd}\right)\left(R_{efgh}R^{efgh}\right)$

Acknowledgments

I would like to thank Thomas Bäckdahl, José M. Martín-García, and Leo Stein for useful discussions and their contributions to the *xAct* mailing list, and Cyril Pitrou for his suggestion to use *xPert* for computing metric variations. Furthermore, I thank Andrea Campoleoni, Massimo Taronna, and Pan Kessel for their feedback on initial versions of *xTras*.

References

[1] José M. Martín-García, et al. xAct: efficient tensor computer algebra for Mathematica, 2002–2013, http://xact.es/.
[2] José M. Martín-García, xPerm: fast index canonicalization for tensor computer algebra, Comput. Phys. Commun. 179 (2008) 597–603.
[3] José M. Martín-García, xTensor: fast abstract tensor computer algebra, 2002–2013, http://xact.es/xTensor/.
[4] R. Portugal, An algorithm to simplify tensor expressions, Comput. Phys. Commun. 115 (1998) 215–230.
[5] R. Portugal, Algorithmic simplification of tensor expressions, J. Phys. A: Math. Gen. 32 (1999) 7779–7789.
[6] Leon R.U. Manssur, Renato Portugal, B.F. Svaiter, Group-theoretic approach for symbolic tensor manipulation, Internat. J. Modern Phys. C 13 (2002) 859–879.
[7] David Brizuela, José M. Martín-García, Guillermo A. Mena Marugán, xPert: computer algebra for metric perturbation theory, Gen. Relativity Gravitation 41 (2009) 2415–2431.
[8] David Brizuela, José M. Martín-García, Guillermo A. Mena Marugán, Harmonics: tensor spherical harmonics in Mathematica, 2006–2013, http://xact.es/Harmonics/.
[9] José M. Martín-García, Leon R.U. Manssur, Renato Portugal, The Invar tensor package, Comput. Phys. Commun. 176 (2007) 640–648.
[10] José M. Martín-García, David Yllanes, Renato Portugal, The Invar tensor package: differential invariants of Riemann, Comput. Phys. Commun. 179 (2008) 586–590.
[11] Alfonso García-Parrado, José M. Martín-García, Spinors: a mathematica package for doing spinor calculus in general relativity, Comput. Phys. Commun. 183 (2012) 2214–2225.
[12] Thomas Bäckdahl, SymManipulator: symmetrized tensor expressions, 2011–2013, http://xact.es/SymManipulator/.
[13] Cyril Pitrou, Xavier Roy, Obinna Umeh, xPand: an algorithm for perturbing homogeneous cosmologies, Classical Quantum Gravity 30 (2013) 165002.
[14] Kasper Peeters, Introducing Cadabra: A Symbolic Computer Algebra System for Field Theory Problems, 2007.
[15] Leo Brewin, A brief introduction to cadabra: a tool for tensor computations in general relativity, Comput. Phys. Commun. 181 (2010) 489–498.
[16] David Yllanes, José M. Martín-García, xCoba: general component tensor computer algebra, 2005–2013, http://xact.es/xCoba/.
[17] Eugene M. Luks, Permutation groups and polynomial-time computation, 11, 1993, pp. 139–175.
[18] Michael B. Green, Kasper Peeters, Christian Stahn, Superfield integrals in high dimensions, J. High Energy Phys. (2005) 0508:093.
[19] P. Etingof, O. Golberg, S. Hensel, T. Liu, A. Schwendner, D. Vaintrob, E. Yudovina, Introduction to Representation Theory, 2009.