xTensorRefGuide

José M. Martín–García

IEM, CSIC, Madrid, Spain
jmm@iem.cfmac.csic.es
http://metric.iem.csic.es/Martin–Garcia/xAct/

This the Reference Guide of the package `xTensor`` , now in version 0.9.5. It is a quick recollection of all commands in the package, with their mutual relations and links to individual help pages. There are no examples: see the notebook `xTensorDoc.nb` for an introduction to the system. There are no formulas: see the LaTeX document `xTensorMaths` for the mathematics underlying the system.

Please report errors, omissions, suggestions or comments to the author. Any kind of help is welcome!

0. Loading

The package `xTensor`` is installed under the directory xAct/ containing all `xAct`` packages. This directory can be installed anywhere and loaded into *Mathematica* giving the full path of installation. However, there are two recommended places for installation of add–ons in *Mathematica*:

- For a single–user installation use:
  - Linux: $HOME/.Mathematica/Applications/
  - Windows: C:\Documents and Settings\USER\Program Data\Mathematica\Applications\
  - Mac: /Users/USER/Library/Mathematica/Applications/
- For a system–wide installation use:
  - Linux: /usr/share/Mathematica/Applications/
  - Windows: C:\Documents and Settings\All Users\Program Data\Mathematica\Applications\
  - Mac: /Library/Mathematica/Applications/

Using one of these directories there is no need to configure any path.

The loaded version of `xTensor`` is contained in the global variable `$Version`. `xTensor`` itself loads the package `xPerm`` of manipulations of large groups of permutations. The minimum version of `xPerm`` required is given by the variable `$xPermVersionExpected`. The loaded version of `xPerm`` is given by `xPerm`$Version`.

If there are error messages during the loading of `xTensor``, it is possible to locate the origin of the error by setting `$ReadingVerbose=True` before reading the package. By default that variable is not set.

1. Symbols and types

■ 1.1. Type information: symbols

There are three primitive types of values in *Mathematica*: symbols (head `Symbol`), strings (head `String`) and num– bers (heads `Integer`, `Rational`, `Real` and `Complex`). Unfortunatlely it is not possible to define new primitive types. Tensors and other types of values must be composite types. What follows in this section refers to tensors, but can also be applied to other `xTensor`` types of values, to be listed below.

Information in *Mathematica* is associated to symbols only (not to strings, numbers or composite expressions as a whole). In `xTensor`` we take the following important decision: information on a tensor will be associated to a symbol identify– ing that tensor. This has two important consequences:

– Tensors are identified using symbols, and not strings.
– We cannot have two different tensors identified by the same symbol, to avoid conflicting information.

This decision has also two important advantages:

– Information on a tensor is only used by *Mathematica* when the tensor appears in the expression being evalu– ated.
– At any time we can collect all the information known about a tensor, using `Information` (the `?` command).

There is a harsh limitation in *Mathematica*: an expression can be associated to a symbol if and only if the symbol is present in the expression at levels 0 or 1, but no deeper. This leads us to introduce a second important decision: symbols with some `xTensor`` type will always appear in the composite expression at level 0; in other words, the symbol identifying a tensor will be the head of the tensor, and so on: we shall use A[...] rather than, for example, the more natural notation Tensor[A][...] suggested by Maeder.

It could seem reasonable to use contexts to separate Tensor‘A from Manifold‘A or Index‘A. This simply means using longer names for the objects defined. We could use as well TensorA, ManifoldA, IndexA, or perhaps TenA, ManiA, IndA. In `xTensor`` we do not force any particular solution, leaving the decision to the user. The only general recom– mendation is using long names for tensors (like MaxwellF for the electromagnetic Faraday tensor) and short names (a, b, C, etc.) for abstract indices.

It could also seem reasonable to define tensors as abstract types, instead of fixing a particular structure from the very beginning. However, this would be slow for pattern matching. We shall simply try to write code having the abstract model in mind.

■ 1.2. Valid symbols. Attributes

Copied from the *Mathematica* Reference Guide (A.1.2): The name of a symbol must be a sequence of letters, letter–like forms and digits, not starting with a digit. `xTensor`` adds a few more restrictions on the symbols that can be used to identify tensors and so on. These restrictions are checked by the *xCore* function `ValidateSymbol`, called by all `DefType` commands:

1. The symbol is not numeric (checked with `NumericQ`).
2. The symbol does not have values (checked with `ValueQ`).
3. The symbol does not have a `Locked` attribute.
4. The symbol is not already used by `xTensor``, `xPerm``, `xCore`` or `ExpressionManipulation``.
5. The symbol is not protected, readprotected or used by *Mathematica.*

There is an exception to restriction 5: the capitals C, D, K, N, O are used by *Mathematica* but are accepted as valid symbols for indices and overloaded (that is, without changing their context), issuing a warning message. The capitals E and I are numeric and cannot be used.

Once a symbol is used to identify an object, it cannot be used to identify another object. We use the function `ValidateSymbolInSession` to check whether a symbol is currently being used or not. This function is called by all `DefType` commands.

All `DefType` commands have the option `ProtectNewSymbol`, whose default value is given by the global variable `$ProtectNewSymbols` (initialized to `False`), which allows the user to protect the defined symbol right after all its properties have been assigned. This is a security feature, and, if used, then any protected symbol must be unprotected (with *Mathematica*'s `Unprotect`) before new definitions can be associated to it.

■ 1.3. Type managing

As we said, `xTensor`` implements its own way to deal with symbol types. It is certainly nor ellegant nor efficient, but it is the only way to use upvalues and keep a simple input, at the expense of harder patterns. Currently there are the following 12 symbol types (the mathematical meaning of each type will be explained in detail in the following sections):

Symbol Type	*Q – function*	*Global list*	*Definition*	*Mean*
ConstantSymbol	ConstantSymbolQ	$ConstantSymbols	DefConstantSymbol	Constant with respec
Parameter	ParameterQ	$Parameters	DefParameter	Parametric d
Manifold	ManifoldQ	$Manifolds	DefManifold	Smooth n – di
VBundle	VBundleQ	$VBundles	DefVBundle	Vector b
AbstractIndex	AbstractIndexQ	$AbstractIndices	DefAbstractIndex	Index associate
Tensor	xTensorQ	$Tensors	DefTensor	Tensor field o
CovD	CovDQ	$CovDs	DefCovD	Connection o
Metric	MetricQ	$Metrics	DefMetric	Metric ten
InertHead	InertHeadQ	$InertHeads	DefInertHead	Wrapper fo
ScalarFunction	ScalarFunctionQ	$ScalarFunctions	DefScalarFunction	Scalar functio
Basis	BasisQ	$Bases	DefBasis	Frame of ve
Chart	ChartQ	$Charts	DefChart	Coordinate chart

Each type has an associated Q–function to identify the symbol type: each user–defined symbol has an upvalue `True` for the corresponding Q–function, giving `False` on the other Q–functions. The name of the Q–function is always con–structed appending Q to the type name (note the exception of `xTensorQ`, to avoid conflict with *Mathematica*'s `Ten–sorQ`). The list of symbols of each type is contained in a global variable whose name is constructed using a `$` and the plural of the symbol type (note that the plural of Index is Indices, and the plural of Basis is Bases). Objects of the corresponding type are defined (undefined) using `DefType` (`UndefType`) commands, where `Type` must be replaced by the corresponding symbol type. The option `Info` allows us to store some information on the type and nature of the defined symbol. The function `Undef` can undefine any symbol.

Bases and charts are defined and dealt with in the companion package `xCoba``. In particular the functions `DefBasis` and `DefChart` are defined in there. However the type management is done by `xTensor``.

Given an expression, we can find all instances of a given type using the function `FindAllOfType`.

■ 1.4. Relations among symbols

There are mathematical objects that can only be defined if other objects have already been defined before. For example defining a scalar field *T* requires the previously defined manifold *M* where it lives. We shall say the symbol *T* is a "visitor" of the symbol *M*, which itself will be called a "host" of *T*. The lists of visitors associated to a symbol is given by the functions `VisitorsOf` and `HostsOf`. A symbol can only be removed when the list of its visitors is empty.

Some objects are automatically defined. For example the tangent bundle of a manifold is automatically defined when the manifold is defined. We say that the tangent bundle is a "servant" of the manifold, and that the manifold is the "master" of the tangent bundle. The list of servants associated to a symbol is given by the function `ServantsOf`. The master of a symbol is given by the function `MasterOf`. The master of a symbol is specified at definition time using the option `Master` of the `DefType` commands.

Visitors of a symbol frequently have that symbol in their own name. For instance, the `Riemann` tensor of the covariant derivative *CD* is called by default *RiemannCD*. This is controlled through the function `GiveSymbol`, which in that example would be called as `GiveSymbol[Riemann,` *CD*], and has explicit instructions on how to proceed in each case.

2. Generalized indices

2.1. Introduction

The main objective of `xTensor`` is the manipulation of indexed objects. From the mathematical point of view we shall always use the notation of abstract indices for abstract expressions (see Penrose & Rindler, or Wald), where the indices denote the type and symmetries of a tensor, and not its components in a given frame (we use a different type of indices for components). This notation is very general and powerful, though sometimes cumbersome. From the computational point of view, however, we need a more general concept of index, with several properties:

1) We define the concept of *generalized index* as any expression found at an *index−slot*. Currently index−slots are those with lower case letters in

tensor[a,b,c]
covd[a][...]
`Bracket`[a][..., ...]

and the indexed arguments of the inert−heads, to be explained later, but nothing else. At those index−slots we can put anything, but the system has been already trained to manipulate five types of expressions, which we now review in detail. This is called the index *type* (these are logic types, not actual symbol types):

− `AIndex`: abstract indices: a, −b, d$101
− `BIndex`: basis indices: {a, polar}, {−b, −polar}
− `CIndex`: component indices: {0, polar}, {2, −polar}
− `DIndex`: directions: `Dir`[vector]
− `LIndex`: labels: `LI`[hello]

They all give `True` when the function `GIndexQ` is applied, and `False` otherwise. There is a sixth type of index, not accepted by `GIndexQ`:

− `PIndex`: patterns with head `Blank`, `Pattern` or `PatternTest`, but no other pattern head

It is possible to introduce new index types but then you would need to specify which values they have for the following index properties (contact JMM to discuss why a new type of index is required):

2) All indices have a *character*, which can be `Down` (*covariant)* or `Up` (*contravariant)*. Nameless patterns like _ or *a_* do not have a well−defined character. However, by convention, they are treated by the formatting routines as if they were contravariant. The character can be detected with the functions `UpIndexQ` and `DownIndexQ`. They both always give `False` on all patterns. Any index can be made contravariant using the function `UpIndex` and covariant using `DownIndex`. The character of any index can be reversed using `ChangeIndex`.

3) Contractible indices are those which obey the Einstein convention and are detected by the function `EIndexQ`. Actual pairs of indices are detected with `PairQ`. Currently only abstract indices and basis indices belong to this special type (though in the future we might consider having non−contracted basis indices). They have a *state*, which can be `Free` or `Dummy` (aka *contracted*). Non−contractible indices are said to have state `Blocked` and can be detected with the function `BlockedQ`, which always gives opposite answers to `EIndexQ`.

4) Apart from those three ‘public´ properties (type, character and state), `xTensor`` uses a fourth property internally: the *metric−state* of an index, saying whether an index has been moved with a metric with respect to the character of the corresponding slot at definition time.

5) To avoid index collisions we use the (*Mathematica* recommended) method of unique variables, having indices like a$123. This is useful but produces ugly expressions. To hide away those "dollar−indices" use the function `ScreenDollarIndices`, either explicitly or by setting `$PrePrint` or `$Post`.

6) Finally, we shall distinguish between indices on a complex vbundle and "conjugated indices" on the complex conjugated vbundle.

As a general recommendation, manipulation of indices must be done using mathematical commands to do that, and not tinkering directly with the indices.

■ 2.2. Abstract indices

Abstract indices are labels that indicate tensorial slots (i. e. contraction with vectors or covectors). In principle we should have indices a, b, c, ..., contravariant indices Up[a], Up[b], Up[c], ..., and covariant indices Down[a], Down[b], Down[c], ... To simplify the input/output of tensors we represent both indices and contravariant indices as a, b, c, ... and covariant indices as –a, –b, –c, ... This is the simplest choice, but the treatment of patterns becomes harder because we loose the symmetry between upindices and downindices (the former being atoms and the latter being composite expres–sions).

The symbol type associated to abstract indices is `AbstractIndex`. (This is a *symbol type* and therefore an expres–sion like –a cannot have this symbol type. See below.) The list of all currently defined abstract indices is given by the global variable `$AbstractIndices`. All of them have associated upvalues `True` for the function `AbstractIn–dexQ`. Abstract indices are defined using the function `DefAbstractIndex` and undefined using `UndefAbstract–Index`. Defining an abstract index just involves checking the validity of the symbol and registering the corresponding upvalue for the Q–function. This should never be done directly by the user. These functions are made public only for consistency of the symbol type management.

Do not confuse the function `AbstractIndexQ`, which gives `True` only on symbols defined as abstract indices (like a, b, ...), and the function `AIndexQ`, which gives `True` on all abstract indices(a, –b, a$125, ...).

We may consider in the future to allow for more general (composite) abstract indices, but currently this is not possible.

In `xTensor`` abstract indices are always associated to vector bundles through the function `VBundleOfIndex`, which is sort of inverse of `IndicesOfVBundle`. The association to manifolds is implicit through their tangent vector bundles.

A simple way to generate a list of abstract indices is provided by the function `IndexRange`.

■ 2.3. Basis indices and component indices

Almost everything related to bases and components is done by the twin package `xCoba``. However, the internal manipulation of basis indices is already prepared in advance in `xTensor``, to avoid a slow process of overloading of functions when loading `xCoba``. Following Schouten, Dodson & Poston, and Penrose & Rindler, basis–indices contain the information of the basis they belong to. This is called the *marked index* notation. That avoids defining different indices for different bases, but makes basis–indices a bit cumbersome: {*a*, *basis*} where *a* is an abstract index. There is no type associated to basis–indices because they are always composite structures.

The function `BIndexQ` checks whether a given expression is a valid basis–index. It has a second argument to check whether the basis index belongs to a given basis or to a given vbundle.

Component indices are defined in parallel with basis indices {*a*, *basis*} but *a* is now an integer, one of those c–numbers ("c" from component and coordinate; no intended connection with Quantum Mechanics) defined with the basis. The function `CIndexQ` checks whether a given expression is a valid component–index. It also has a second argument to check whether the component index belongs to a given basis or to a given vbundle. The index {1, *basis*} is contravariant and the index {1, –*basis*} is covariant. The integer number can be positive, negative or zero because the character is stored as the sign of the basis, and not of the integer itself. Beware that this was different in pre–0.8 versions of `xTen–sor``.

The function `BCIndexQ` checks whether an index is a basis–index or a component–index. The function `ABIndexQ` checks whether an index is an abstract–index or a basis–index, currently the only two types of indices which are contractible.

■ 2.4. Directional indices

Directional indices represent the mathematical notation that sees tensorial slots as slots for vectors and covectors. Because it is seldom used, we want to avoid `xTensor`` checking continuosly if a given index is directional. Hence we introduce the head `Dir` to pinpoint the directional indices.

The vector argument of a directional index has its own index, which is not an index of the whole expression. The index must belong to the correct vector bundle, but the name of the index itself is irrelevant; it is some kind of dummy index; we call it an ultraindex.

The function `DIndexQ` checks whether a given expression is a valid directional−index. Vectors can be contracted to `Dir` expressions using `ContractDir` and separated using `SeparateDir`.

■ 2.5. Label indices

There are indices which are not associated to vector bundles. We call them *labels* or *label−indices*. An example could be the *l, m* labels of the spherical harmonics. Because they are used only every now and then, we denote them with a special head: `LI`. An `LI` expression can have any internal structure; in particular they can have several elements. It is possible to associate a character for them using `LI`[a] and −`LI`[a] (`LI`[−a] is not interpreted as a "covariant label") but they are defined as not obeying the Einstein convention (blocked indices).

The function `LIndexQ` checks whether a given expression is a valid label−index.

■ 2.6. Patterns

At index−slots we can also find patterns for g−indices (even patterns for patterns). Not all patterns are allowed: only those with head `Blank`, `Pattern` or `PatternTest`. The function `PIndexQ` validates the allowed patterns. The function `PatternIndex` constructs patterns of the required form.

For a description of the patterns to be used in rules, see Section 6 below.

■ 2.7. Finding indices

All indices of an expression, including patterns, can be extracted using the function `FindIndices`. This function has attribute `HoldFirst` to allow it getting the indices of input expressions, before they start to evaluate. `FindIndices` always returns a list of indices with head `IndexList`, to avoid confusion with the notation for basis and component indices. `FindIndices[0]` returns `IndexList[AnyIndices]`. `FindIndices` works recursively, checking the heads of the elements of expressions, and complaining when it finds an unknown head. New "known" heads can be added to the list `$FindIndicesAcceptedHeads`.

When searching for the indices of a tensor, covariant derivative of a tensor or a tensor product we check that none of the indices are repeated with the internal function `CheckRepeated`. When searching for the indices of a sum of tensor expressions, or a list, equation or rule of them, we check homogeneity of free indices with the internal function `Check-Homogeneity`.

Three related functions, based on `FindIndices`, are `FindFreeIndices`, `FindDummyIndices` and `Find-BlockedIndices`. They give disjoint lists of indices. The second one returns only the up–member of the pairs of dummies.

A very friendly driver for `FindIndices` is `IndicesOf`. The general syntax is `IndicesOf`[*selectors*][*expr*], where *selectors* are one or several of the following:

– `Free`: free indices
– `Dummy`: dummy indices
– `Blocked`: blocked indices
– `Up`: contravariant indices
– `Down`: covariant indices
– `AIndex`: abstract indices
– `BIndex`: basis indices
– `CIndex`: component indices
– `DIndex`: directional indices
– `LIndex`: label indices:
– *vbundle*: indices of the given vbundle
– *basis*: indices of the given basis
– *tensor*: indices on the given tensor
– *covd*: indices on the given covariant derivative
– `Basis`[*basis*]: (both) indices on `Basis` objects of the given basis
– `Not`[any of the previous]: complement of the previous

A sequence of several selectors represents the `And` of all selectors (smaller result). A list of several selectors represents the `Or` of all selectors (bigger result).

An alternative and completely independent way of looking for indices, very useful for recursive algorithms of index contraction, is the function `IsIndexOf`.

■ 2.8. Sorting indices

The canonicalization of an indexed object essentially entails to a reordering of the indices according to the symmetries of the object and a predefined ordering for the indices. The function `IndexSort` returns the preferred order for a list of indices. With the function `SetIndexSortPriorities` we can decide which particular order we want to have. Possible priorities are the strings `"up"/"down"`, `"free"/"dummy"`, `"lexicographic"/"antilexicographic"`, `"positional"/"antipositional"`.

Simple functions derived from `IndexSort` are `IndexOrderedQ` and `DisorderedPairQ`.

■ 2.9. Replacing indices

The basic function for index replacement in a generic expression is `ReplaceIndex`. Every valid index can be changed by any other thing, not necessarily an index (though that would inmediately produce many errors). `ReplaceIndex` has attribute `HoldFirst`. The syntax is `ReplaceIndex`[*expr*, *rules*], where the *rules* are of the form *index*–>*newindex*. The rule *a*–>*b* and –*a*–>–*b* are considered independent, and both must be specified if that is what we need.

Derived functions are:

– `ReplaceDummies`: replacement of all dummies in an expression by indices in a given list, or by new dollar–indices. Indices belonging to different vbundles are not mixed up. In computations with intensive generation of dollar–indices the memory of the computer could be filled after a while and the global variable `$ComputeNewDummies` has been introduced to avoid this. There is also a private function `RemoveDollarIndices`.

– `SameDummies`: returns an expression minimizing the number of different dummies used in different terms.

– `PermuteIndices`: replace indices in an expression as given by a permutation or a linear combination of them.

– `SplitIndex`: returns a list of expressions where a given free index has been respectively replaced by each of a list of indices. This is useful to expand component ranges or to expand sums of vbundles.

– `TraceDummy`: converts an expression with a dummy pair into a sum of expressions with different dummy pairs. Log messages can be controlled with `$TraceDummyVerbose`.

3. Formatting of indexed objects

Formatting in versions 0.7and 0.8 was rather limited. Version 0.9 has introduced cut–and–paste, but still no editing of the tensor expressions. This section will be much expanded in future versions.

Most symbols in `xTensor`` can be formatted in `StandardForm`. The output form of a symbol is a string "<output>" chosen at definition time using the option `PrintAs`, which defines an upvalue `PrintAs[`*symbol*`]="`<output>`"`. The value of the option can be directly the string "<output>" or a function which returns the string when applied on symbol. Formatting of a type of objects (or of all allowed objects to be formatted) can be turned on/off using the functions `xTensorFormStart[`*type*`]` / `xTensorFormStop[`*type*`]`.

Those symbols automatically generated with `GiveSymbol` have a parallel function to generate their output strings, called `GiveOutputString`.
Indices are always formatted if a `PrintAs` upvalue has been given for them. Note that the formatting of indices is not given at definition time; it must be explicitly set as upvalues for `PrintAs`.

Basis and component indices are formatted using a color associated to the corresponding basis. Components are num–bered, but for coordinate systems it is possible to use the name of the corresponding coordinate (`CIndexForm`, `$CIndexForm`).

Covariant derivatives and their indices can be formatted in `StandardForm` in two possible ways, stored as values in the global variable `$CovDFormat`: `"Prefix"` and `"Postfix"`, with obvious meanings. The associated symbols to be used in each of those cases are stored in `SymbolOfCovD` for each covariant derivative.

4. Mathematical entities

Elementary mathematical objects in `xTensor``.

■ 4.1. Constant–symbols

Symbols defined with type `ConstantSymbol` represent constants with respect to all kinds of derivatives. In particular, they are given attribute `Constant`.

Constant–symbols are defined with `DefConstantSymbol` and undefined with `UndefConstantSymbol`. Possible options at definition time are `Dagger` and those generic for all Def–commands: `Info`, `Master`, `ProtectNewSymbol` and `PrintAs`.

The list of all currently defined constant–symbols is stored in the global variable `$ConstantSymbols`.

Any symbol defined as a constant–symbol is given a `True` upvalue for the function `ConstantSymbolQ`, which is defined as `False` on any other input.

A constant–symbol is a particular kind of constant. A constant is either a constant–symbol, a numeric symbol or a number. We use the function `ConstantQ` to check that something is a constant. Do not confuse `ConstantQ` and `ConstantSymbolQ`. Replacing constant–symbols by constants is safe. For instance, there is no problem in using a rule like `Mass->2`.

■ 4.2. Parameters

Symbols defined with type `Parameter` represent real parameters with some undefined range of values. Essentially they will be used as dependencies of other objects (tensors, for instance), and we will be able to take parametric derivatives of expressions (see `ParamD` below).

Parameters are defined with `DefParameter` and undefined with `UndefParameter`. There are no particular options at definition time, apart from those generic for all `DefType` commands: `Info`, `Master`, `ProtectNewSymbol` and `PrintAs`.

We do not expect parameters to be master symbols (i.e. have servants). However they can have objects, those objects which depend on the parameter. A parameter cannot be undefined if it has objects.

The list of all currently defined parameters is stored in the global variable `$Parameters`.

Any symbol defined as a parameter is given a `True` upvalue for the function `ParameterQ`, which is defined as `False` on any other output.

The parameter dependencies of a generic expression expr are obtained applying `ParametersOf` on *expr*. This is just a call to `DependenciesOf` on *expr* and then a selection (using `Select` and `ParameterQ`) of the parameters. Parameter dependencies of a tensor are obtained, following a parallel path, using the private function `ParametersOfTensor`. Note that the latter function expects a symbol (the tensor head), but the former expects a generic expression.

■ 4.3. Manifolds and vector bundles

■ 4.3.1. Manifolds

A symbol *manifold* with type `Manifold` represents a smooth, differentiable manifold of fixed dimension.

Manifolds are defined with `DefManifold` and undefined with `UndefManifold`. The syntax for definition of *manifold* is
`DefManifold[`*manifold*, *dim*, *indices*`]`, where:
manifold is the symbol to be defined,
dim is a nonnegative integer or a constant−symbol, and
indices is the list of abstract indices associated to the tangent vbundle of manifold.

The list of all currently defined manifolds is stored in the global variable `$Manifolds`. All of them have associated upvalues `True` for the function `ManifoldQ`, which is defined as `False` on any other input.

The dimensionality of the manifold is stored as an upvalue of *manifold* for the function `DimOfManifold`. 0−dim and 1−dim manifolds have not been fully implemented yet.

■ 4.3.2. Product manifolds

Given several manifolds it is possible to define their "product−manifold" structure. This is done using again `DefManifold`, but now the second argument is a list of the (previously defined) submanifolds. The list of submanifolds is stored as an upvalue for the function `SubmanifoldsOfManifold`, and give `True` when asked by `SubmanifoldQ`. The list of all defined product−manifolds is stored in the global variable `$ProductManifolds`, which is a subset of `$Manifolds`.

■ 4.3.3. Dependencies

The manifold dependencies of a generic expression *expr* are obtained using applying `ManifoldsOf` on *expr*. This is just a call to `DependenciesOf` on *expr* and then a selection (using `Select` and `ManifoldQ`) of the parameters. Manifold dependencies of a tensor are obtained, following a parallel path, using the private function `ManifoldsOfTensor`. Note that the latter function expects a symbol (the tensor head), but the former expects a generic expression.

Dependencies are specially relevant for derivatives, where it is always important to know whether a derivative on some expression shares dependencies with that expression. This is computed via the function `DisjointManifoldsQ`.

■ 4.3.4. Vector bundles ("vbundles")

Each manifold has an associated vbundle (its tangent bundle) with the same dimension, whose name is formed by joining the symbol `Tangent` and the name of the manifold, and which is stored as an upvalue for the function `TangentBundleOfManifold`.

However, in `xTensor`` it is possible to define more general ("inner") vbundles, of the type that are used in gauge theories. As usual in `xTensor`` we only worry about local properties, and therefore all our vbundles are considered products of an inner vector space and the base manifold. This is a new type of symbol:

A symbol *vbundle* with type `VBundle` represents a smooth vector bundle of fixed dimension.

Vbundles are defined with `DefVBundle` and undefined with `UndefVBundle`. The syntax for definition of *vbundle* is `DefVBundle[`*vbundle*, *manifold, dim*, *indices*`]`, where:
vbundle is the symbol to be defined,
manifold is the base manifold of *vbundle*,
dim is a nonnegative integer or a constant–symbol, the dimension of the vector space, and
indices is the list of abstract indices associated to *vbundle*.

The list of all currently defined vbundles is stored in the global variable `$VBundles`. All of them have associated upvalues `True` for the function `VBundleQ`, which is defined as `False` on any other input.

The dimensionality of the vbundle (that is, that of its vector space) is stored as an upvalue of *vbundle* for the function `DimOfVBundle`. 0–dim and 1–dim vbundles have not been implemented yet. The base manifold is stored as an upvalue of the vbundle for the function `BaseOfVBundle`.

■ 4.3.5. Sum vbundles

Given several vbundles it is possible to define their "sum–vbundle" structure. This is done using again `DefVBundle`, but now the third argument is a list of the (previously defined) subvbundles. The list of subvbundles is stored as an upvalue for the function `SubvbundlesOfVBundle`, and give `True` when asked by `SubvbundleQ`. The list of all defined sum–vbundles is stored in the global variable `$SumVBundles`, which is a subset of `$VBundles`.

Dummy pairs in a vbundle can be converted into sums of dummy pairs of its vbundles using the function `TraceProductDummy` (with infix notation `CircleDot`).

■ 4.3.6. Indices and vbundles

The list *indices* is stored as the first element of two in the list `IndicesOfVBundle`. The second element will be the list of indices internally generated when the number of registered indices in not enough. See the function `NewIndexIn`. Indices can be added to the first list (`AddIndices`) or removed from it (`RemoveIndices`), even though the latter is very dangerous because previous expressions are likely to get corrupted if they contain removed indices. We can get any number of abstract indices using the function `GetIndicesOfVBundle`. Unique (dollar–) dummy indices can be generated using `DummyIn`. A list of respective dollar–indices on the subvbundles of a vbundle can be obtained with `SubdummiesIn`.

▪ 4.4. Tensors

▪ 4.4.1. Type

A symbol *tensor* with type `Tensor` represents a smooth tensor field living on some *manifold*. If *manifold* is 0–dim, then *tensor* is actually not a field but an algebraic object.

Tensors are defined with `DefTensor` and undefined with `UndefTensor`. The syntax for definition of *tensor* is `DefTensor[`*tensor*[*indices*], *dependencies*`]` for a tensor without symmetries and `DefTensor[`*tensor*[*indices*], *dependencies*, *symmetry*`]` in general, where:

indices is the a sequence of abstract indices denoting the type of tensor

dependencies is a list (or a single symbol) containing the manifolds *tensor* lives on, and/or parameters it depends upon.

symmetry is a generating set or a strong generating set describing the symmetry properties of the tensor.

The list of all currently defined tensors is stored in the global variable `$Tensors`. All of them have associated upvalues `True` for the function `xTensorQ`, which is defined as `False` on any other input. Note the difference between this function and the *Mathematica* built–in `TensorQ` (new in *Mathematica* 5.0).

■ 4.4.2. Properties and options

The slot structure of the tensor (`SlotsOfTensor`) is stored as `{-M1, M1, M2, ...}`, such that the first slot is a covariant index on manifold M1, the third slot is a contravariant index on M2, etc.

The list *dependencies* of manifolds and/or parameters is stored as an upvalue of *tensor* for the function `DependenciesOfTensor`. The order in the list is irrelevant because it is overwritten using the private function `SortDependencies`, which sorts parameters before manifolds, and uses lexicographic order in both sets. If the tensor has no dependencies then use `{}`. A tensor is always a tensor field on the manifolds corresponding to its indices; that is, we consider that a nonscalar tensor cannot be a constant on a given manifold. This is because we need additional structure to show that a tensor field does not depend on that manifold (for example a vector field, in order to take Lie derivatives). The list of parameter dependencies can be obtained with the private function `ParametersOfTensor`; the list of manifold dependencies can be obtained with the private function `ManifoldsOfTensor`.

Using option `WeightOfTensor` (default value 0) we can define tensorial densities, defined as a linear combination of bases. The weight is stored as an upvalue of *tensor* for the function `WeightOfTensor`. Tensor densities are represented in output using Ashtekar's notation: weight n positive (negative) is represented adding n tildes above (below) the name of the tensor. The tildes are colored according to the bases they represent. The weight of a generic expression is computed using `WeightOf`.

The *symmetry* description of the tensor is assumed to be a generating set (head `GenSet`) or a strong generating set (head `StrongGenSet`) of the symmetry group of slot permutations. Simple cases can be constructed using the functions `Symmetric`, `Antisymmetric` and `RiemannSymmetry`. The information is always stored as a strong generating set being an upvalue of *tensor* for the function `SymmetryGroupOfTensor`. Handling of multiterm symmetries through Young tableaux is under development, and will be stored using the function `SymmetryTableauxOfTensor`. Any perm notation can be used on input, but it will always be changed to `Cycles` notation on numeric slots. Different slots belonging to the same orbit of the symmetry group must have the same slot structure (same manifold and same up/down character) at definition time, but this can be overwritten using the option `ForceSymmetries` (default is `False`).

Under complex conjugation (with `Dagger`) tensors can behave in four different ways. They can be
 – `Real` (the default): the tensor is invariant under complex conjugation.
 – `Imaginary`: the tensor changes sign under complex conjugation.
 – `Hermitian`: the tensor is invariant under simultaneous complex conjugation of indices and exchange of indices between a vbundle and its conjugate.
 – `Complex`: generic case.
This is given through the option `Dagger` and stored as an upvalue of `Dagger` for the tensor.

Apart from the usual `DefType` options (`Info`, `Master`, `ProtectNewSymbol` and `PrintAs`), there are other options:
 `FrobeniusQ`: not functional now
 `OrthogonalTo`: vectors to which the tensor is orthogonal
 `ProjectedWith`: projectors leaving the tensor invariant
 `TensorID`: information on how to compute components
 `VanishingQ`: if a tensor vanishes, a delayed definition of the form *tensor*`[___]:=0` is set.

■ 4.4.3. Special tensors

There are three heads with special tensorial meaning. Two of them are the `delta` and the "generalized delta" `Gdelta` tensors:

`delta`[*a*, −*b*] is the identity tensor on the vbundle of its indices. There is no difference between this object and `delta`[−*b*, *a*] and actually `delta` is defined as symmetric (though not orderless), even though its indices are always staggered. If both its indices have the same character, then `delta` is inmediately converted into the (first) metric tensor of the vbundle of its indices. If one of its indices is a basis index then `delta` is converted into `Basis`, which is formally equivalent, but with the Orderless attribute.

`Gdelta`[*a1,...,an, −b1,...,−bn*] is the generalized delta tensor on any vector bundle. Its first half of indices is antisymmetric, and so it is the second half, independently. The function `ExpandGdelta` converts the `Gdelta` tensor into a linear combination of products of *n* `delta`'s, as given by a determinant.

The other one has been created only for convenience: `Zero` represents the 0 tensor, for any indices.

■ 4.4.4. Name generation

There are a number of tensors which are automatically defined. They are associated to other objects (vbundles, bases, connections, etc.) and their names are constructed using the function `GiveSymbol`, with syntax `GiveSymbol[`*tensor*, *object*`]`, where *tensor* is one of the following reserved words:

`epsilon`	*metric*	totally antisymmetric tensor covariantly constant with respect to the metric
`Christoffel`	*covd*	Christoffel symbol associated to the connection covd and the fiducial `PD`
`AChristoffel`	*covd*	internal Christoffel symbol associated to the connection covd
`Torsion`	*covd*	torsion tensor associated to the connection *covd*
`Riemann`	*covd*	curvature tensor associated to the connection *covd*
`FRiemann`	*covd*	internal curvature tensor associated to the connection *covd*
`Ricci`	*covd*	Ricci tensor associated to the connection *covd*
`TFRicci`	*metric−covd*	Trace−free Ricci tensor associated to the connection *metric−covd*
`RicciScalar`	*metric−covd*	Ricci scalar associated to the metric connection *metric−covd*
`Einstein`	*metric−covd*	Einstein tensor associated to the metric connection *metric−covd*
`Weyl`	*metric−covd*	Weyl tensor associated to the metric connection *metric−covd*
`Projector`	*induc−metric*	Projector tensor on a codimension−1 surface with induced metric *induc−metric*
`ExtrinsicK`	*induc−metric*	Extrinsic curvature tensor of a codimension−1 surface with induced metric *induc−metric*
`Acceleration`	*vector*	Acceleration of *vector*

The symbol `Christoffel` can be also associated to two different covds.

■ 4.5. Covariant derivatives

■ 4.5.1. Internal format for covariant derivatives

The format for a covariant derivative is `CD[-a][expr]`. The double pair of brackets roughly follows *Mathematica*'s structure for derivatives: `f'[x]` is represented as `Derivative[1][f][x]`, separating the derivative operator `Derivative[1]` from the object being differentiated. However we do not want to distinguish between the differentiation operator (`D`) and the internal representation for a derivative (`Derivative`). Nor we can separate the abstract object being differentiated (`f`) from the field variable or the indices (that would require a complete change of philosophy in `xTensor`` `). All this means that higher derivatives must be stored as nested derivatives: `CD[-a][CD[-b][expr]]`. This is natural with respect to the multiple possible derivative operators, but has several drawbacks:

1. Many pairs of brackets are needed. This is partially alleviated using the prefix notation `CD[-a]@CD[-b]@expr`.
2. Even though `CD[-a][T[-b]]` is a tensor, as well as `T[-b]`, the notations are very different. Every function acting on tensorial inputs must be prepared to receive a covariant derivative.
3. Rules for `T[-a]` will be replaced in derivatives of `T[-a]`. This can be considered a drawback or an advantage, depending on the case.
4. Because of the depth restriction in *Mathematica*, rules for second–or–higher derivatives of `T[-a]` cannot be upvalues for `T`.

This could be extended to other simpler formats in future versions if required by a large fractions of the users of *xTensor*.

■ 4.5.2. Type

A symbol *covd* with type `CovD` represents a smooth connection or covariant derivative living on some *manifold*, and acting on tensor fields with indices in some *vbundle* having that *manifold* as base. If *manifold* is 0–dim then connections are not allowed. Rather than working with a single derivative operator and Christoffel symbols for different derivatives, we define different derivative operators for different connections, following Wald.

Covariant derivatives are defined with `DefCovD` and undefined with `UndefCovD`. The syntax for definition of *covd* is `DefCovD`[*covd*[−*a*], *symbol*] or `DefCovD`[*covd*[−*a*], *vbundle, symbol*], where:

a is an index on a tangent vbundle which identifies the manifold where the covariant derivative lives.

symbol is a list containing two strings: the first / last one gives the `"Postfix"` / `"Prefix"` output in Standard–Form.

vbundle is the vbundle on which the covariant derivative acts. If not given it is assumed to be the (tangent) vbundle of the index *a*.

The list of all currently defined covariant derivatives is stored in the global variable `$CovDs`. All of them have associated upvalues `True` for the function `CovDQ`, which is defined as `False` on any other input. Do not confuse `CovDQ` with `FirstDerQ`, to be explained below.

■ 4.5.2. Properties and options

The manifold on which the connection lives is stored as an upvalue for *covd* of the function `ManifoldOfCovD`. It is currently not possible in *xTensor* to define a parameter dependency for a connection. The list of *vbundles* on which the connection acts is stored as an upvalue for *covd* of the function `VBundlesOfCovD` (the first one is always the tangent vbundle of that *manifold)*. The *symbol* of *covd* is stored as an upvalue of `SymbolOfCovD`. Which of the `"Postfix"` / `"Prefix"` formats is used is decided by the global variable `$CovDFormat`. A number of options are possible at definition time, concerning whether the covariant derivative has or not `Torsion`, `Curvature` or derives `FromMetric`, among other. The information related to these options is stored as upvalues for *covd* of the functions `TorsionQ`, `CurvatureQ` and `MetricOfCovD`, respectively. Another option is `CurvatureRelations`, which determines whether the contractions of the `Riemann` tensor must be replaced by the `Ricci` tensor, and the contractions of `Ricci` by the `RicciScalar`. When this option is set to False then those relations must be explicitly implemented using the function `ContractCurvature`. Another option is `ExtendedFrom`, which allows defining a derivative acting on an inner vbundle and whose action on the corresponding tangent vbundle is exactly that of a previously defined covariant derivative. Finally, Levi–Civita connections can be modified to act on densities associated to a given basis, specified through the option `WeightedWithBasis`, typically used through `DefMetric`.

All connections are assumed to be real, and so there is no need to use the `Dagger` option.

As usual, there are also the options `ProtectNewSymbol`, `Master` and `Info`.

■ 4.5.3. The fiducial derivative

The space of covariant derivatives is an affine space, with no preferred point. It is customary, however, to choose as origin for this space a particular but unspecified ordinary derivative. We will call it `PD`. It has zero torsion and zero curvature. By convention, it is the origin for Christoffel tensors, as we will see below. These "partial derivatives" are not automatically commuted (see below how to do it). The canonicalization process has the option `CommutePDs` (default `True`) to control this issue. There is no metric associated to `PD` by default, but the user can add it.

■ 4.5.4. Associated tensors

A number of tensors are automatically associated to each derivative: `Torsion`, `Riemann`, `Ricci` and the `Christoffel` tensor relating it to the fiducial `PD`. If the derivative comes from a metric then we have additionally: `TFRicci`, `RicciScalar`, `Einstein`, `Weyl`. If the derivative acts on an inner vbundle then the tensors `FRiemann` and `AChristoffel` are also automatically associated. What follows is valid for all those tensors but we use the example of `Torsion`: the torsion tensor associated to a connection *CD* is denoted with the symbol *TorsionCD* and this is done by calling the function `GiveSymbol[Torsion,` *CD*`]`, whose behaviour can be freely chosen. We can also use `Torsion[`*CD*`][`*inds*`]`, which is automatically converted into `GiveSymbol[Torsion,` *CD*`][`*inds*`]` and hence, by default, into *TorsionCD*[*inds*].

The `Christoffel` tensor is special because it is actually associated to two covariant derivatives and can be denoted as `Christoffel[`*CD1*, *CD2*`][`*inds*`]` (antisymmetric in the derivatives), which we call the Christoffel of *CD1* from *CD2*. By `Christoffel[`*CD*`][`*inds*`]` we understand `Christoffel[`*CD*, `PD][`*inds*`]`. The expression `Christoffel[`*CD1*, *CD2*`][`*inds*`]` is automatically converted into the tensor *ChristoffelCD1CD2*[*inds*] if {*CD1*, *CD2*} are sorted lexicographically or into –*ChristoffelCD2CD1*[*inds*] in the opposite case. The tensor is defined during the process if it did not exist before. The derivative `PD` is always sorted last. Any Christoffel tensor (of *CD1* from *CD2*) can be rewritten using the function `BreakChristoffel` as the sum of two Christoffel tensors, the first of *CD1* from *CD3* and the second of *CD3* from *CD2*, for any *CD3* on the same manifold as *CD1* and *CD2*.

Not any two covariant derivatives can be related via a Christoffel tensor. This is only possible if the derivatives are "compatible" (checked with the private function `CompatibleCovDsQ`): they act on the same base manifold and they share the vbundles or at least one of them does not act on any inner vbundle.

We need several commands to change the order of derivatives acting on a tensor. The command `CommuteCovDs` exchanges the order of two (equal) derivatives identified by the user through their respective indices. `SortCovDs` brings the derivative operators to canonical order of their indices. Commands `SortCovDsStart` and `SortCovDsStop` turn on and off, respectively, the automatization of the function `SortCovDs`. The canonicalization routines commute equal covds on scalars by default, but this behaviour can be changed using `$CommuteCovDsOnScalars`.

There is a number of commands which change some tensors into equivalent expressions. These are: `ChangeCovD` (previously known as `CovDToChristoffel`), `ChangeTorsion` (previously known as `TorsionToChristoffel`), `ChangeCurvature` (previously known as `RiemannToChristoffel`), and the pairs `RiemannToWeyl` / `WeylToRiemman`, `RicciToEinstein` / `EinsteinToRicci` and `RicciToTFRicci` / `TFRicciToRicci`. When there is a metric we also have the pair `ChristoffelToGradMetric` / `GradMetricToChristoffel` (the first of the pair was previously known as `ChristoffelToMetric`).

Finally, there are several variables controlling convention signs: `$RiemannSign`, `$RicciSign`, `$TorsionSign`. We will find some more of these later.

■ 4.5.5. Dependencies

Each derivative lives on a given manifold. On objects not having that manifold as a dependency the derivative gives zero. Checking this fact takes some time and *xTensor* does not do it automatically. The function `CheckZeroDerivative` is in charge of that, and its action can be automatized using `CheckZeroDerivativeStart` and `CheckZeroDerivativeStop`. The global variable `$CheckZeroDerivativeVerbose` turns on/off the messages reporting when `CheckZeroDerivative` is being used and on which object.

■ 4.6. Metrics

■ 4.6.1. Type

A symbol *metric* with type `Metric` represents a smooth 2–symmetric field living on some *manifold*.

Metrics are defined with `DefMetric` and undefined with `UndefMetric`. The syntax for definition of *metric* is `DefMetric`[*signdet, metric*[−*a*, −*b*], *covd, covdsymbols*], where:

signdet gives information on the signature of the metric: it is either 0, 1, −1 or a list of integers {pluses, minuses, zeroes}

−*a*, −*b* are covariant abstract indices on the vbundle where *metric* is being defined

covd is the Levi–Civita connection associated to *metric*, with symbols *covdsymbols*

The list of all currently defined metrics is stored in the global variable `$Metrics`. All of them have associated upval– ues `True` for the function `MetricQ`, which is defined as `False` on any other input.

■ 4.6.2. Properties and options

A metric is always defined on a given vbundle (that of its abstract indices at definition time), which is stored as an upvalue for the function `VBundleOfMetric`. However, a vbundle can have several metrics (stored in the function `MetricsOfVBundle`). A vbundle with at least one metric gives `True` under the function `MetricEndowedQ`, and `False` if it has not got any metric. If there are several metrics only the first one will be used to raise and lower indices; all other metrics are called "frozen" and do not have all the expected properties for the first–metric. In particular, the inverse of a frozen metric *frozen*[−*a*, −*b*] is not *frozen*[a, b] (which is actually *g*[*a, c*] *g*[*b, d*] *frozen*[−*c*, −*d*], with *g* being the first–metric), but is defined as *Invfrozen*[*a, b*], using the head `Inv`.

Every metric has a unique torsionless covariant derivative, called its Levi–Civita connection and stored as an upvalue of the metric for the function `CovDOfMetric`. Covariant derivatives can be or not associated to a metric, and this is stored in the function `MetricOfCovD`, which returns `Null` if the connection does not derive from a metric. If this associated connection is flat then we say that the metric is flat, and this can be specified at definition time with the Boolean option `FlatMetric`, whose value is stored as an upvalue for `FlatMetricQ`.

The only invariant information associated to a metric is its signature, defined as a list of +1's, −1's and 0's, which can be specified at definition time as the first argument of `DefMetric`, and is stored as an upvalue for the function `Signa– tureOfMetric`. The product of those numbers is the sign of the determinant of the metric (in any basis), and is given by the function `SignDetOfMetric`.

Associated to the metric we have the `epsilon` tensor, the uniquely defined (up to global constant) totally antisymmet– ric tensor. Its global sign is given by the variable `$epsilonSign`. The curvature tensors associated to the metric are actually those associated to its Levi–Civita connection (`Riemann` and `Ricci`). Having a metric gives us a number of additional curvature tensors: `RicciScalar`, `Einstein`, `TFRicci` and `Weyl`.

■ 4.6.3. Product metrics

Given a number of vbundles with their respective metrics, it is possible to define a block–form "product–metric" of them using the syntax `DefProductMetric`[*metric*[−*a*, −*b*], { {*vbundle1*, *scalar1*[]}, {*vbundle2*, *scalar2*[]}, ... }, *covd*, *covdsymbol*], where:

metric[−*a*, −*b*] is the metric being defined, with indices on a previously defined sum–vbundle
scalar1[] is a scalar field on the base manifolds of *vbundle2, ...*, but not of *vbundle1*
covd is the Levi–Civita connection of *metric*
covdsymbol is the pair of symbols used for *covd* in `StandardForm`

The defined metric is, essentially,

scalar1[]^2 *metric1*[., .] + *scalar2*[]^2 *metric2*[., .] + ...

with *metric1* being the first–metric of *vbundle1*, etc. The scalars are stored using the function `MetricScalar`. The list of defined all product–metrics is given by the global variable `$ProductMetrics` and it is always a subset of the metrics in `$Metrics`.

The function `ExpandProductMetric` converts objects associated to the product–metric into combinations of the objects associated to the sub–metrics.

■ 4.6.4. Induced metrics

Given a metric field *g* and a surface–orthogonal (see `FrobeniusQ`) vector field *v*, it is possible to induce a metric *h* on that surface. This structure can be defined using the option `InducedFrom` of `DefMetric`. The association with the vector field *v* is stored in `VectorOfInducedMetric`. It is only possible to associate induced metrics to the first–metric of a vbundle. Induced metrics are never considered frozen metrics.

Working with induced metrics is based on the use of four objects:

– The projector onto the hypersuface. There is an inert–head acting as a formal projector, and this is constructed using the head `Projector` and the name of the induced metric. The projector *h*[*a*, −*b*] can be introduce using `ProjectWith`[*h*], and can be converted into a tensorial expression *g*[*a*, −*b*] − *v*[*a*] *v*[−*b*] / *norm* (where *norm* is the norm of *v* in the metric *g*) using `ProjectorToMetric` and its inverse `MetricToProjector`. Any tensor can be decomposed in parts which are parallel or orthogonal to *v* using `InducedDecomposition`.

– As with any other metric, *h* has an associated Levi–Civita connection, but in this case this operator is a true derivative only when acting on tensors orthogonal to *v*. This connection can be expressed in terms of the connection of *g* and projectors using `ProjectDerivative`.

– The `Acceleration` vector of *v*. Its sign is given by a convention stored in the variable `$Acceleration–Sign`.

– The extrinsic curvature of *h*, formed with the symbol `ExtrinsicK`. Its sign is given by a convention stored in the variable `$ExtrinsicKSign`. It is possible to change from the extrinsic curvature tensor to derivatives of *v* using the function `ExtrinsicKToGradNormal` and its inverse `GradNormalToExtrinsicK`.

■ 4.6.5. Metric contraction

Given the metric *g*[−*a*, −*b*] and the vector field *T*[*b*], it is customary to denote the expression *g*[−*a*, −*b*]*T*[*b*] as *T*[−*a*], and the change from the former to the latter is called "contraction" in `xTensor`‘. Contractions with a metric are never automatic (compare with the automatic contraction of `delta`), and are inforced using the command `ContractMet–ric`. The inverse operation is implemented in `SeparateMetric`. When there are several metrics on the same vbun–dle, only the first–metric can be contracted and separated. All other metrics are called "frozen".

There are two options for `ContractMetric`: `OverDerivatives` and `AllowUpperDerivatives`, with obvious meanings.

A second form of separating metrics is using the function `SetCharacters`, which introduces metric factors to change the characters of the indices of one or several tensors.

■ 4.7. Bases and charts

It is not always enough to arrive at an abstract tensor field expression. Very often we need to introduce a basis of vectors, or even a chart, in order to get the final result of a computation. `xTensor`` has been designed as a manipulator of abstract expressions, and therefore we need to implement bases and charts in an abstract way as well. This is implemented in the companion package `xCoba``, but the types `Basis` and Chart have been already implemented here:

A symbol *basis* with type `Basis` represents a basis of vector fields on a given vbundle. The list of all currently defined bases is stored in the global variable `$Bases`. All of them have associated upvalues `True` for the function `BasisQ`, which is defined as `False` on any other input. The functions `DefBasis` and `UndefBasis` are defined in the package `xCoba``.

In parallel, a symbol *chart* with type `Chart` represents a smooth chart on a given manifold. The list of all currently defined charts is stored in the global variable `$Charts`. All of them have associated upvalues `True` for the function `ChartQ`, which is defined as `False` on any other input. The functions `DefChart` and `UndefChart` are defined in the package `xCoba``.

■ 4.8. Other derivatives

Apart from covariant derivatives there are other types of derivations currently supported by `xTensor``:

Lie derivatives are denoted using the head `LieD`. The general syntax is `LieD`[*vector*][*expr*] where *vector* is any tensorial expression with a single upper abstract free index. That index is not relevant except for its character and associated vbundle; we call it an *ultraindex*. Lie derivatives can be expanded using a covariant derivative with the function `LieDToCovD`.

Lie brackets are denoted using the head `Bracket`. The general syntax is `Bracket`[*a*][*vect1*, *vect2*] where *vect1* and *vect2* are two contravariant vector fields with free ultraindices. The index of the resulting vector field is *a* and not the ultraindex. Lie brackets can be expanded using a covariant derivative with the function `BracketToCovD`.

There are two kinds of parametric derivatives in `xTensor``, for historical reasons. The operator *Mathematica* builtin `OverDot` has been overloaded as a derivative with respect to an arbitrary parameter. Every tensor field is assumed to depend on that parameter, unless stated otherwise. The recommended parametric derivative is, however, `ParamD`, with syntax `ParamD`[*par1*, *par2*, ...][*expr*] where *par1, par2, ...* are parameters (defined with `DefParameter`) with respect we differentiate.

A variational derivative `VarD` is planned for future versions.

The command `FirstDerQ` identifies single derivatives: it gives `True` on expressions of the form *covd*[−*a*], `LieD`[*v*], `OverDot` or `ParamD`[*par*], and `False` otherwise (in particular on multiple parametric derivatives).

A variational derivative `VarD` is planned for future versions.

5. Input Expressions

Composite mathematical objects in `xTensor``.

■ 5.1. Sum of tensors

There is no special "tensor addition" command. We use the Plus head in *Mathematica* because this allows us to use many builtins which already know how to handle Plus (in particular the simplification algorithms). Any input expres–sion in xTensor` is assumed to be a sum of terms and most algorithms are threaded over those terms in such a way that each term is manipulated independently.

The use of Plus is not a restriction in the sense that it has all expected properties of a sum of tensors. The only problem might be the attribute Orderless (implementing commutativity) because we cannot control the order in which the terms are placed.

■ 5.2. Tensor product

There is no special "tensor product" command. We use the Times head in *Mathematica* because this allows us to use many builtins which already know how to handle Times (in particular the simplication algorithms). Any term expres–sion in xTensor` is assumed to be a product of factors. A tensor product can be considered as a single tensor and many algorithms in xTensor` use this idea.

The product of several tensors can be separated into monomials which do not share dummy indices. This can be done with the function BreakInMonomials, which introduces the (inert–) head Monomial.

The use of Times is a restriction in the sense that it is a commutative product (implemented throught its Orderless attribute). There is no natural anticommutative product in *Mathematica* and xTensor` does not try to introduce it. Apart from that, Times is perfectly general because the abstract indices keep track of the structure of the expression.

■ 5.3. Scalars and the Scalar head

A monomial with no free indices is a scalar field, and it is often convenient to mark scalar fields as such. We do this using the head Scalar (which could be, but has not been, defined as an inert–head). The main property of a Scalar expression is that it hides the indices inside from the computations, so that xTensor` treats a Scalar expression as a block, like it would do with a truly elementary scalar field. For instance, dummy indices can be repeated across different Scalar expression in the same product.

To separate Scalar expressions use the function PutScalar (which is essentially a call to BreakInMonomials), and to remove the Scalar head use NoScalar. Sometimes Scalar expressions can be further subdivided, and this is achieved with the function BreakScalars.

The function ScalarQ detects scalars, that is expressions with no free indices (recall that only indices of types A and B can be free indices; blocked indices are never free indices). An expression with head Scalar is certainly a scalar, but constant–symbols, parameters or any other expression without free indices are also scalars. Similar functions, detecting expressions with just a single free index are UpVectorQ and DownVectorQ.

■ 5.4. Inert heads

We call inert–head a symbol *h* such that *h*[*expr*] has the same tensorial character as *expr* (same indices with same characters, and same symmetries), even though *h* is not assumed to be linear in general. Such a symbol will be given type `InertHead`.

Inert–heads are defined with `DefInertHead` and undefined with `UndefInertHead`. There are two particular option at definition: `LinearQ`, which states whether the inert–head is linear or not (value stored as an upvalue for the function with same name), and `ContractThrough`, which gives a list of metrics (and/or `delta`) which can be contracted through the inert–head (value stored as an upvalue for the function `ContractThroughQ`). Additionally, we have the generic options for all `DefType` commands: `ProtectNewSymbol`, `Info`, `Master` and `PrintAs`.

The list of all currently defined inert–heads is stored in the global variable `$InertHeads`.

Any symbol defined as an inert–head is given a `True` upvalue for the function `InertHeadQ`, which is defined as `False` on any other input.

■ 5.5. Scalar functions

In `xTensor`` there is a second way in which we can have tensors as arguments of functions: scalar functions of scalar arguments are allowed, and they must be registered before being used. Those functions will be called scalar–functions and their symbols will be given type `ScalarFunction`.

Scalar–functions are defined with `DefScalarFunction` and undefined with `UndefScalarFunction`. There are no particular options at definition time, apart from some of those generic for all `DefType` commands: `ProtectNew-Symbol`, `Info`, `Master` and `PrintAs` (the latter one is currently not in use).

A second argument at definition time denotes the number of arguments of the scalar–function (default is 1). That number is stored as an upvalue for the function `NumberOfArguments`.

Scalar–functions cannot be master symbols (i.e. cannot have servants). They cannot have objects either.

The list of all currently defined scalar–functions is stored in the global variable `$ScalarFunctions`, which is initialized to `{Exp, Log, Sin, Cos, Tan, Csc, Sec, Cot, Power, Factorial}`.

Any symbol defined as a scalar–function is given a `True` upvalue for the function `ScalarFunctionQ`, which is defined as `False` on any other input.

There is no special formatting rules for scalar–functions.

The arguments of a scalar–function can be wrapped with the `Scalar` head, but in general this is not necessary.

■ 5.6. Complex conjugation

`xTensor`` has its own complex–conjugation operator, called `Dagger`, to avoid overloading the *Mathematica* builtin `Conjugate`. All input expressions have a definite behaviour under the `Dagger` operation, and this is controlled using `Dagger` as an option in the `DefType` commands. Possible values are `Real` (usually the default), `Complex`, `Imaginary`, `Hermitian` and `Antihermitian`. Special definitions are introduced for the object being defined as specified by the value of that option. The function `DaggerQ` returns `True` on *expr* if `Dagger`[*expr*] is different from *expr*.

Indices can also carry information on the complex properties of the object they belong to. Conjugation of indices is performed by the function `DaggerIndex`. Tensors with equal numbers of indices on a vbundle and its conjugate can be `Hermitian`. Their conjugation properties are implemented through the function `TransposeDagger`.

Finally, by default the conjugated symbol to a given symbol (tensor or index) is formed by adding a character to the original symbol. This character is stored in the global variable `$DaggerCharacter`, and initially is the dagger character "†".

■ 5.7. Validation

The function `Validate` checks the syntax of an expression in `xTensor``. When doing a computation there are some checks but not many, to save time. In those cases in which the error can be localized in a particular subexpression of the whole expression, that subexpression is returned wrapped with the inert–head `ERROR` (printed in red in `StandardForm`).

6. Rules and definitions

Rules among tensor expressions. There are two levels to consider: 1) ensuring syntactically correct rules and 2) having flexible ways of producing rules.

■ 6.1. Indicial rules

Given the simple structure of our tensor expressions, it is tempting to construct simple rules to replace tensors by other tensor expressions. However that would inmediately produce syntactic errors, like repeated indices (see examples of this in `xTensorDoc.nb`). `xTensor`` generalizes the four main rule constructs to work with indexed expressions, with new names having the prefix `Index`:

`Rule`	`IndexRule` (infix notation `RightTeeArrow`)
`RuleDelayed`	`IndexRuleDelayed`
`Set`	`IndexSet` (infix notation `DoubleRightTee`)
`SetDelayed`	`IndexSetDelayed`

■ 6.2. MakeRule

The function `MakeRule` offers a large flexibility in constructing tensor rules and their equivalents under certain changes, as controlled by its options. The syntax is either `MakeRule[{`*lhs, rhs*`}`, *options*`]` or `MakeRule[{`*lhs, rhs, conditions*`}`, *options*`]` if we want to add conditions (head `Condition`) to the final rules. Possible options are:

- `PatternIndices`: indices to be converted into patterns
- `TestIndices`: whether vbundle of indices must be checked
- `MetricOn`: indices on which the metric must be used
- `UseSymmetries`: whether symmetries of tensors must be used or not
- `ContractMetrics`: whether to contract metric factors on the rhs
- `Verbose`: report on the internal progress

■ 6.3. Automatic rules

The rules produced by `MakeRule` or any other rules can be converted into permanent definitions (like those produced by `Set`) using the function `AutomaticRules`. This function works like the `TagSet` family, deciding whether the rule must be defined as a downvalue or an upvalue for a given symbol. If none of those is possible then the rule is appended to the list `$Rules`, which must be imposed explicitly by the user.

7. Manipulation of input

■ 7.1. Symmetry

Every product of tensors or tensorial expressions has a well defined symmetry under permutations of its indices, and this can be obtained with the function `SymmetryOf`. For convenience, apart from the permutation group describing the symmetry, this function returns the original expression with indices numbered, so that it is clear which indices the permutations are referring to. For example, for a tensor `Rie` with the symmetries of a Riemann tensor, the symmetry returned by `SymmetryOf` would be this expression with head `Symmetry`:

```
Symmetry[4, Rie●1●2●3●4, {●1 → G, ●2 → F, ●3 → D, ●4 → K},
 StrongGenSet[{1, 2, 3}, GenSet[Cycles[{1, 3}, {2, 4}], -Cycles[{1, 2}], -Cycles[{3, 4}]]]]
```

The symmetry group is written in strong generating set notation, and its permutations are written in cyclic notation. For explanation of these and other concepts in permutation group theory see the documentation for the companion package `xPerm``.

The symmetry of a product of tensors is computed from the symmetries of the individual tensors (stored in `Symmetry-GroupOfTensor`) and taking into account the possibility of permuting equal subexpressions. When there are deriva- tives involved the computation is more complicated and we need to know whether the derivatives commute, or whether it is possible to permute indices with different characters. The options `CommutePDs` and `ConstantMetric` of `SymmetryOf` help in controlling these points. The global variable `$CommuteCovDsOnScalars` turns on and off the commutativity of symmetric covariant derivatives on scalar fields.

■ 7.2. Canonicalization and simplification

The main part of a computer algebra system is the canonicalizer, the algorithm in chart of bringing any expression to its canonical form. In `xTensor`` the canonicalizer is implemented in a single command, called `ToCanonical`, by far the most sophisticated algorithm of the whole system. Its action is composed of three steps:

1) On a sum of terms we first apply the function `SameDummies` to minimize the number of different dummy indices. Then we map ToCanonical over individual terms, such that each of them is canonicalized independently.

2) Terms (generically products of different objects) are sorted according to a number of criteria. This is done by the function `xSort`. This function works in three internal steps, corresponding to three respective internal (private) functions:

2.1) `Identify`: Dismantle the expression adding symbols characterizing each of its parts

2.2) `MarkBlocked`: mark those subexpressions with only blocked indices; they do not require canonicalization

2.3) `ObjectSort`: sorts the different parts of the expression taking into account their properties. The global variable `$CommuteFreeIndices` controls the ordering of equivalent objects with free indices.

3) Once the term has been sorted, it can be considered as a single tensor with indices and symmetry as given by `SymmetryOf`. Then we "only" have to call the algorithms for canonicalization of permutations in single and double cosets which have been developed by R. Portugal and his collaborators. These algorithms have been encoded in the companion package `xPerm`` and constitute the hardest part of the canonicalization process. `xPerm`` offers two different (but equivalent) encodings of the algorithm: a pure–*Mathematica* code `CanonicalPerm` and a mixed–C–*Mathematica* code `MathLinkCanonicalPerm`, which is much faster but is not available for all platforms (see the documentation of `xPerm`` for details). Which of the two is used is chosen through the option `MathLink` of `ToCanonical`. (The name of the option comes from the fact the *MathLink* protocol is used the link the C and *Mathematica* parts of the code.) By default `ToCanonical` returns only the canonical expression, but the option `GivePerm` returns both the canonical expression and the corresponding canonical permutation. The option `Notation` controls how permutations are handled internally.

Apart from those three (permutation–related), there are three more options for `ToCanonical`. One of them reports information on the progress of the canonicalization process: `Verbose`. (There are also the options `xPermVerbose` and `TimeVerbose` to get information and timings on the actual permutation–canonicalization process from `xPerm``.) Then there is the option `UseMetricOnVBundle`, which gives a list of vbundles on which the metric can be used to raise and lower the indices. The final issue is that of canonicalization of derivatives: when there is a metric and a derivative which is not compatible with that metric, the system changes to the internal function `ToCanonicalDers`, which handles canonicalization much more carefully, but also much more slowly. That change can be avoided by switching off the global variable `$MixedDers`. This new algorithm usually produces lots of Christoffel tensor because it changes internally from the "offending" derivative to the Levi–Civita connection of the metric. It is possible to convert automatically those Christoffel tensors into derivatives of the metric using the option `ExpandChristoffel`.

Finally, there is the function `Simplification`, which is simply a combination of `ToCanonical` and then call to `Simplify`.

■ 7.3. Imposing symmetries

Given an expression *expr* and a symmetry group *G* the function `ImposeSymmetry`[*expr, inds, G*] constructs the linear combination of all index–permutations of *expr* corresponding to the elements of the group *G* applied on the indices *inds* of *expr*, in particular taking *expr* to be the expression corresponding to the identity element. The result is always divided by the order of *G* (the number of elements). Special derived functions for special groups of permutations are `Symmetrize`, `Antisymmetrize`, `PairSymmetrize` and `PairAntisymmetrize`, with obvious meanings.

We can also handle symmetry operations involving a metric: the function `STFPart` returns the symmetric trace–free part of an expression with respect to a given metric.

More ambitious, but still restricted to the case of a single vbundle, are the functions `ChangeFreeIndices`, which changes the free abstract indices of an expression to those given by the user, and the function `EqualExpressionsQ`, which checks whether two expressions are the same apart from symmetries and permutations of indices.

■ 7.4. Collecting terms

There are three simple functions which help in manipulating tensor expressions. These three functions are currently very simple and will be improved in future versions:
`IndexCoefficient`[*expr, form*] returns the coefficient of *form* in *expr*.
`IndexCollect`[*expr*, *form*, *function*] imitates the action of `Collect` but allowing for indexed expressions in *form*.
`IndexSolve`[*equation, tensor*] solves *equation* for the given *tensor* in very simple cases: *tensor* has only free indices

■ 7.5. Acting on particular subexpressions

In `xTensor`` there are no special functions or arguments to act at particular positions of an expression. This is because *Mathematica* already offers lots of different possibilities to act on arbitrary positions in different ways. See for example the functions `Map`, `MapAt`, `MapAll`, `MapIndexed`, etc. However, it is sometimes difficult to know in which position a given subexpression is, and for this an other similar purposes the functions `ColorPositionsOfPattern` and `ColorTerms` are really useful. These two have been constructed using the functionality of the great package ExpressionManipulation` by David J.M. Park Jr., Ted Ersek (C) 1999–2007.

8. List of commands

`ABIndexQ`

`AbstractIndex`

`AbstractIndexQ`

`$AbstractIndices`

`Acceleration`

`$AccelerationSign`

`AChristoffel`

`AddIndices`

AIndex

AIndexQ

AllowUpperDerivatives

Antihermitian

Antisymmetrize

AnyDependencies

AnyIndices

AutomaticRules

BaseOfVBundle

$Bases

Basis

BasisQ

BCIndexQ

BIndex

BIndexQ

Blocked

BlockedQ

Bracket

BracketToCovD

BreakChristoffel

BreakInMonomials

BreakScalars

CDIndexQ

ChangeCovD

ChangeCurvature

ChangeFreeIndices

ChangeIndex

ChangeTorsion

Chart

ChartQ

$Charts

CheckZeroDerivative

CheckZeroDerivativeStart

CheckZeroDerivativeStop

$CheckZeroDerivativeVerbose

Christoffel

ChristoffelToGradMetric

ChristoffelToMetric

CIndex

CIndexForm

$CIndexForm

CIndexQ

CircleDot

ColorPositionsOfPattern

ColorTerms

CommuteCovDs

$CommuteCovDsOnScalars

$CommuteFreeIndices

CommutePDs

Complex

$ComputeNewDummies

ConstantMetric

ConstantQ

ConstantSymbol

ConstantSymbolQ

$ConstantSymbols

ContractCurvature

ContractDir

ContractMetric

ContractMetrics

ContractThrough

ContractThroughQ

CovD

$CovDFormat

CovDOfMetric

CovDQ

$CovDs

CovDToChristoffel

Curvature

CurvatureQ

CurvatureRelations

Dagger

$DaggerCharacter

DaggerIndex

DaggerQ

DefAbstractIndex

DefConstantSymbol

DefCovD

DefInertHead

DefManifold

DefMetric

DefParameter

DefProductMetric

DefScalarFunction

DefTensor

DefVBundle

delta

DependenciesOf

DependenciesOfTensor

DimOfManifold

DimOfVBundle

DIndex

DIndexQ

Dir

Disclaimer

DisjointManifoldsQ

DisorderedPairQ

DoubleRightTee

Down

DownIndex

DownIndexQ

DownVectorQ

Dummy

DummyIn

EIndexQ

Einstein

EinsteinToRicci

epsilon

$epsilonSign

EqualExpressionsQ

ERROR

ExpandChristoffel

ExpandGdelta

ExpandProductMetric

ExtendedFrom

ExtrinsicK

$ExtrinsicKSign

ExtrinsicKToGradNormal

FindAllOfType

FindBlockedIndices

FindDummyIndices

FindFreeIndices

FindIndices

$FindIndicesAcceptedHeads

FirstDerQ

FlatMetric

FlatMetricQ

ForceSymmetries

Free

FRiemann

FrobeniusQ

FromMetric

Gdelta

GetIndicesOfVBundle

GIndexQ

GiveOutputString

GivePerm

GiveSymbol

GradMetricToChristoffel

GradNormalToExtrinsicK

Hermitian

HostsOf

Imaginary

ImposeSymmetry

IndexCoefficient

IndexCollect

IndexForm

IndexList

IndexOrderedQ

IndexRange

IndexRule

IndexRuleDelayed

IndexSet

IndexSetDelayed

IndexSolve

IndexSort

IndicesOf

IndicesOfVBundle

InducedDecomposition

InducedFrom

InertHead

InertHeadQ

$InertHeads

Info

Inv

IsIndexOf

Labels

LI

LieD

LieDToCovD

LIndex

LIndexQ

LinearQ

MakeRule

Manifold

ManifoldOfCovD

ManifoldQ

$Manifolds

ManifoldsOf

Master

MasterOf

MathLink

Metric

MetricEndowedQ

MetricOfCovD

MetricOn

MetricQ

$Metrics

MetricScalar

MetricsOfVBundle

MetricToProjector

$MixedDers

Monomial

NewIndexIn

NoScalar

Notation

NumberOfArguments

VisitorsOf

OrthogonalTo

OverDerivatives

OverDot

PairAntisymmetrize

PairQ

PairSymmetrize

ParamD

Parameter

ParameterQ

$Parameters

ParametersOf

PatternIndex

PatternIndices

PD

PermuteIndices

PIndex

PIndexQ

PrintAs

$ProductManifolds

$ProductMetrics

ProjectDerivative

ProjectedWith

ProtectNewSymbol

$ProtectNewSymbols

Projector

ProjectorToMetric

ProjectWith

PutScalar

$ReadingVerbose

Real

RemoveIndices

ReplaceDummies

ReplaceIndex

Ricci

RicciScalar

$RicciSign

RicciToEinstein

RicciToTFRicci

Riemann

$RiemannSign

RiemannToChristoffel

RiemannToWeyl

RightTeeArrow

$Rules

SameDummies

Scalar

ScalarFunction

ScalarFunctionQ

$ScalarFunctions

ScalarQ

ScreenDollarIndices

SeparateDir

SeparateMetric

ServantsOf

SetCharacters

SetIndexSortPriorities

SetOrthogonal

SignatureOfMetric

SignDetOfMetric

Simplification

SlotsOfTensor

SortCovDs

SortCovDsStart

SortCovDsStop

SplitIndex

STFPart

SubdummiesIn

SubmanifoldQ

SubmanifoldsOfManifold

SubvbundleQ

SubvbundlesOfVBundle

$SumVBundles

SupermanifoldsOfManifold

SymbolOfCovD

Symmetrize

Symmetry

SymmetryGroupOfTensor

SymmetryTableauxOfTensor

SymmetryOf

Tangent

TangentBundleOfManifold

Tensor

TensorID

$Tensors

TestIndices

TFRicci

TFRicciToRicci

ToCanonical

Torsion

TorsionQ

$TorsionSign

TorsionToChristoffel

TraceDummy

$TraceDummyVerbose

TraceProductDummy

TransposeDagger

Undef

UndefAbstractIndex

UndefConstantSymbol

UndefCovD

UndefInertHead

UndefManifold

UndefMetric

UndefParameter

UndefScalarFunction

UndefTensor

UndefVBundle

Up

UpIndex

UpIndexQ

UpVectorQ

UseMetricOnVBundle

UseSymmetries

Validate

ValidateSymbolInSession

VanishingQ

VarD

VBundle

VBundleQ

VBundleOfIndex

$VBundles

VBundlesOfCovD

VBundleOfMetric

VectorOfInducedMetric

Verbose

$Version

WeightedWithBasis

WeightOf

WeightOfTensor

Weyl

WeylToRiemann

$xPermVersionExpected

xSort

xTensorFormStart

xTensorFormStop

xTensorQ

Zero

9. Possible changes in the system

This is a list of possible changes I'm considering for future versions. Each of them is discussed in a different notebook. If you have any comment or suggestion on one of those changes (either in favour or against it), please edit the corresponding notebook and send it to me:

Options and their associated functions

Notations for multiple covariant derivatives