Ricci

A Mathematica package for doing tensor calculations in differential geometry

User's Manual

VERSION 1.2

BY JOHN M. LEE
ASSISTED BY DALE LEAR AND JOHN ROTH

Ricci

A Mathematica package for doing tensor calculations
in differential geometry

User's Manual

Version 1.2

By John M. Lee
assisted by Dale Lear and John Roth

Development of this software was supported in part
by NSF grant DMS-9101832

John M. Lee
Department of Mathematics, GN-50
University of Washington
Seattle, WA 98195
Internet: `lee@math.washington.edu`
Fax: 206-543-0397

Contents

1 Introduction

1.1 Overview

Ricci is a Mathematica package for doing symbolic tensor computations that arise in differential geometry. It supports:

- Tensor expressions with and without indices
- The Einstein summation convention
- Correct manipulation of dummy indices
- Mathematical notation with upper and lower indices
- Automatic calculation of covariant derivatives
- Automatic application of tensor symmetries
- Riemannian metrics and curvatures
- Differential forms
- Any number of vector bundles with user-defined characteristics
- Names of indices indicate which bundles they refer to
- Complex bundles and tensors
- Conjugation indicated by barred indices
- Connections with and without torsion

Ricci is named after Gregorio Ricci-Curbastro (1853-1925), who invented the tensor calculus (what M. Spivak calls "the debauch of indices").

This manual describes the capabilities and functions provided by Ricci. To use Ricci and this manual, you should be familiar with Mathematica, and with the basic objects of differential geometry (manifolds, vector bundles, tensors, connections, and covariant derivatives). Chapter 8 contains a complete reference list of all the Ricci commands, functions, and global variables. Most of the important Ricci commands are described in some detail in the main text, but there are a few things that are explained only in Chapter 8.

I would like to express my gratitude to my collaborators on this project: Dale Lear, who wrote the first version of the software and contributed uncountably many expert design suggestions; John Roth, who reworked some of the most difficult sections of code; Jay Coskey and Pm Weizenbaum, who contributed invaluable editorial assistance with this manual; the National Science Foundation and the University of Washington, who provided generous financial support for the programming effort; and all those mathematicians who tried out early versions of this software and contributed suggestions for improvement.

1.2 Obtaining and using Ricci

Ricci requires Mathematica version 2.0 or greater. The source takes approximately 278K bytes of disk storage, including about 49K bytes of on-line documentation. I have tested the package on a DEC Alpha system, where it runs reasonably fast and seems to require about 6 or 7 megabytes of memory. I don't have any idea how it will run on other systems, but the Alpha is a pretty fast machine, so I expect that Ricci will be very slow on some platforms.

The source files for Ricci are available to the public free of charge, either via the World-Wide Web from `http://www.math.washington.edu/~lee/Ricci/` or by anonymous ftp from `math.washington.edu`, in directory `pub/Ricci`.

You'll need to download the Ricci source file `Ricci.m` and put it in a directory that is accessible to Mathematica. (You may need to change the value of Mathematica's `$Path` variable in your initialization file to make sure that Mathematica can find the Ricci files—see the documentation for the version of Mathematica that you're using.) Once you've successfully transferred all the Ricci files to your own system, start up Mathematica and load the Ricci package by typing `<<Ricci.m`. Once you've loaded Ricci into Mathematica, you can type `?name` for information about any Ricci function or command.

If you have questions about using Ricci, suggestions for improvement, or a problem that you think may be caused by a bug in the program, please contact the author `<lee@math.washington.edu>`.

1.3 A brief look at Ricci

To give you a quick idea what typical Ricci input and output look like, here are a couple of examples. Suppose alpha and beta are 1-tensors. Ricci can manipulate tensor products and wedge products:

```
In[4]:= TensorProduct[alpha,beta]

Out[4]= alpha (X) beta

In[5]:= Wedge[alpha,beta]

Out[5]= alpha ^ beta
```

and exterior derivatives:

```
In[6]:= Extd[%]

Out[6]= d[alpha] ^ beta - d[beta] ^ alpha
```

To express tensor components with indices, you just type the indices in brackets immediately after the tensor name. Lower and upper indices are typed as `L[i]` and `U[i]`, respectively; in output form they appear as subscripts and superscripts. Indices that result from covariant differentiation are typed in a second set of brackets. For example, if `alpha` is a 1-tensor, you indicate the components of `alpha` and its first covariant derivative as follows:

```
In[7]:= alpha [L[i]]

Out[7]= alpha
             i

In[8]:= alpha [L[i]] [L[j]]

Out[8]= alpha
             i; j
```

Ricci always uses the Einstein summation convention: any index that appears as both a lower index and an upper index in the same term is considered to be implicitly summed over, and the metric is used to raise and lower indices. For example, if the metric is named `g`, the components of the inner product of `alpha` and `beta` can be expressed in either of the following ways:

```
In[9]:= alpha[L[k]] beta[U[k]]

                     k
Out[9]= alpha   beta
             k


In[10]:= g[U[i],U[j]] alpha[L[i]] beta[L[j]]

                          i j
Out[10]= alpha   beta   g
              i      j
```

Ricci's simplification commands can recognize the equality of two terms, even when their dummy indices have different names. For example:

```
In[11]:= %9 + %10

                          i j                k
Out[11]= alpha   beta   g     + alpha  beta
              i      j                k

In[12]:= TensorSimplify[%]

                        i
Out[12]= 2 alpha   beta
                i
```

You can take a covariant derivative of this last expression just by putting another index in brackets after it:

```
In[13]:= % [L[j]]

                            i                  i
Out[13]= 2 (alpha      beta     + alpha   beta      )
                 i ;j                  i        ;j
```

The remainder of this manual will introduce you more thoroughly to the capabilities of Ricci.

2 Ricci Basics

There are four kinds of objects used by Ricci: bundles, indices, constants, mathematical functions, and tensors. This chapter describes these objects, and then describes how to construct simple tensor expressions. The last section in the chapter describes how to save your work under Ricci.

2.1 Bundles

In Ricci, the tensors you manipulate are considered to be sections of one or more vector bundles. Before defining tensors, you must define each bundle you will use by calling the `DefineBundle` command. When you define a bundle, you must specify the bundle's name, its dimension, the name of its metric, and the index names you will use to refer to the bundle. You may also specify (either explicitly or by default) whether the bundle is real or complex, the name of the tangent bundle of its underlying manifold, and various properties of the bundle's default metric, connection, and frame.

The simplest case is when there is only one bundle, the tangent bundle of the underlying manifold. For example, the command

```
In[2]:= DefineBundle[ tangent, n, g, {i,j,k} ]
```

defines a real vector bundle called `tangent`, whose dimension is `n`, and whose metric is to be called `g`. The index names `i`, `j`, and `k` refer to this bundle. If any expression requires more than three tangent indices, Ricci generates index names of the form `k1`, `k2`, `k3`, etc. If the bundle is one-dimensional, there can only be one index name.

Every bundle must have a metric, which is a nondegenerate inner product on the fibers. By default, Ricci assumes the metric is positive-definite, but you can override this assumption by adding the option `PositiveDefinite -> False` to your `DefineBundle` call. The metric name you specify in your `DefineBundle` call is automatically defined by Ricci as a symmetric, covariant 2-tensor associated with this bundle.

By default, Ricci assumes all bundles to be real. You may define a complex bundle by including the option `Type -> Complex` in your `DefineBundle` call. For example, suppose you also need a complex two-dimensional bundle called `fiber`. You can define it as follows:

```
In[3]:= DefineBundle[ fiber, 2, h, {a,b,c}, Type -> Complex]
```

This specifies that `fiber`'s metric is to be called `h`, and that the index names `a`, `b`, and `c` refer to `fiber`.

The fifth argument in the example above, `Type -> Complex`, is a typical example of a Ricci option. Like options for built-in Mathematica functions, options for Ricci functions are typed after the required arguments, and are always of the form `optionname -> value`. This option specifies that `fiber` is to be a complex bundle.

Any time you define a complex bundle, you also get its conjugate bundle. In this case the conjugate bundle is referred to as `Conjugate[fiber]`; the barred versions of the indices `a`, `b`, and `c` will refer to this conjugate bundle.

Mathematically, the conjugate bundle is defined abstractly as follows. If V is a complex n-dimensional vector bundle, let $J\colon V \to V$ denote the complex structure map of V—the real-linear endomorphism obtained by multiplying by $i = \sqrt{-1}$. If $\mathbf{C}V$ denotes the complexification of V, i.e., the tensor product (over $\mathbf{R}$) of V with the complex numbers, then J extends naturally to a complex-linear endomorphism of $\mathbf{C}V$. $\mathbf{C}V$ decomposes into a direct sum $\mathbf{C}V = V' \oplus V''$, where V', V'' are the i and $(-i)$-eigenspaces of J, respectively. V' is naturally isomorphic to V itself. By convention, the conjugate bundle is defined by $\overline{V} = V''$.

The metric on a complex bundle is automatically defined as a 2-tensor with Hermitian symmetries. In Ricci, this is implemented as follows. A Hermitian tensor h on a complex bundle is a real, symmetric, complex-bilinear 2-tensor on the direct sum of the bundle with its conjugate, with the additional property that $h_{ab} = h_{\bar{a}\bar{b}} = 0$. Thus the only nonzero components of h are those of the form

$$h_{a\bar{b}} = h_{\bar{b}a} = \overline{h_{b\bar{a}}}.$$

(It is done this way because Ricci's index conventions require that all tensors be considered to be complex-multilinear.) To obtain the usual sesquilinear form on a complex bundle V, you apply the metric to a pair of vectors $x, \overline{y}$, where $x, y \in V$. In Ricci's input form, this inner product would be denoted by `Inner[ x, Conjugate[y] ]`.

For every bundle you define, you must specify, either explicitly or by default, the name of the tangent bundle of the underlying manifold. Ricci needs this information, for example, when generating indices for covariant derivatives. By default, the first bundle you define (or the direct sum of this bundle and its conjugate if the bundle is complex) is assumed to be the underlying tangent bundle for all subsequent bundles. If you want to override this behavior, you can either give a value to the global variable `$DefaultTangentBundle` before defining bundles, or explicitly include the `TangentBundle` option in each call to `DefineBundle`. See the Reference List, Chapter 8, for more details.

Another common type of bundle is a tangent bundle with a Riemannian metric. The `DefineBundle` option `MetricType -> Riemannian` specifies that the bundle is a Riemannian tangent bundle. Riemannian metrics are described in

Section 7.2.

There are other options you can specify in the `DefineBundle` call, such as `FlatConnection`, `TorsionFree`, `OrthonormalFrame`, `CommutingFrame`, `PositiveDefinite`, and `ParallelFrame`. If you decide to change any of these options after the bundle has already been defined, you can call `Declare`. See the Reference List, Chapter 8, for a complete description of `DefineBundle` and `Declare`.

2.2 Indices

The components of tensors are represented by upper and lower indices. Ricci adheres to the Einstein index conventions, as follows. For any vector bundle V, sections of V itself are considered as contravariant 1-tensors, while sections of the dual bundle V^* are considered as covariant 1-tensors. The components of contravariant tensors have upper indices, while the components of covariant tensors have lower indices. Higher-rank tensors may have both upper and lower indices. Indices are distinguished both by their horizontal positions and by their vertical positions; the altitude of each index indicates whether that index is covariant or contravariant. Any index that appears twice in the same term must appear once as an upper index and once as a lower index, and the term is understood to be summed over that index. (Indices associated with one-dimensional bundles are an exception to this rule; see Section 7.1 on one-dimensional bundles below.)

In input form and in Ricci's internal representation, indices have the form `L[name]` for a lower index and `U[name]` for an upper index. Barred indices (for complex bundles only) are typed in input form as `LB[name]` and `UB[name]`; they are represented internally by `L[name[Bar]]` and `U[name[Bar]]`. In output form, upper and lower indices are represented by superscripts and subscripts, with or without bars. This special formatting can be turned off by setting `$TensorFormatting=False`.

Every index name must be associated with a bundle. This association is established either by listing the index name in the call to `DefineBundle`, as in the examples above, or by calling `DefineIndex`. You can create new names without calling `DefineIndex` just by appending digits to an already-defined index name. For example, if index `j` is associated with bundle `x`, then so are `j1`, `j25`, etc.

2.3 Constants

In the Ricci package, a constant is any expression that is constant with respect to covariant differentiation in all directions. You can define a symbol to be a constant by calling `DefineConstant`:

```
In[10]:= DefineConstant[ c ]
```

This defines `c` to be a real constant (the default), and ensures that covariant derivatives of `c`, such as `c[L[i]]`, are interpreted as `0`. To define a complex constant, use

```
In[11]:= DefineConstant[ d, Type -> Complex ]
```

If a constant is real, you may specify that it has other attributes by giving a `Type` option consisting of one or more keywords in a list, such as `Type -> {Positive,Real}`. The complete list of allowable `Type` keywords for constants is `Complex`, `Real`, `Imaginary`, `Positive`, `Negative`, `NonPositive`, `NonNegative`, `Integer`, `Odd`, and `Even`. The `Type` option controls how the constant behaves with respect to conjugation, exponentiation, and logarithms.

After a constant has been defined, you can change its `Type` option by calling `Declare`, as in the following example:

```
In[12]:= Declare[ d, Type -> {NonNegative,Real} ]
```

In the Ricci package, any expression that does not contain explicit tensors is assumed to be a constant for the purposes of covariant differentiation, so in some cases it is not necessary to call `DefineConstant`. However, if you attempt to insert indices into a symbol `c` that has not been defined as either a constant or a tensor, you will get output that looks like `c[L[i],L[j]]`, indicating that Ricci does not know how to interpret `c`. For this reason, as well as to tell Ricci what type of constant `c` is, it is always a good idea to define your constants explicitly.

Note that constants are not given Mathematica's `Constant` attribute, for the following reason. Some future version of Ricci may allow tensors depending on parameters such as `t`; in this case, `t` will be considered a constant from the point of view of covariant differentiation, but may well be non-constant in expressions such as `D[f[t],t]`. You can test whether a tensor expression is constant or not by applying the Ricci function `ConstantQ`.

2.4 Tensors

The most important objects that Ricci uses in calculations are tensors. Ricci can handle tensors of any rank, and associated with any vector bundle or direct sum or tensor product of vector bundles. Scalar functions are represented by 0-tensors; other objects such as vector fields, differential forms, metrics, and curvatures are represented by higher-rank tensors.

Tensors are created by the `DefineTensor` command. For example, to create a 1-tensor named `alpha` you could type:

```
In[6]:= DefineTensor[ alpha, 1 ]
```

This creates a real tensor of rank one, which is associated with the current default tangent bundle, usually the first bundle you defined. All tensors are assumed by default to be covariant; thus the tensor `alpha` defined above can be thought of as a covector field or 1-form. To define a contravariant 1-tensor (i.e., a vector field), you could type:

```
In[7]:= DefineTensor[ v, 1, Variance -> Contravariant ]
```

To create a scalar function, you simply define a tensor of rank 0:

```
In[8]:= DefineTensor[ u, 0, Type -> {NonNegative, Real} ]
```

The `Type` option of `DefineTensor` controls how the tensor behaves with respect to conjugation, exponentiation, and logarithms. For a rank-0 tensor, its value can be either a single keyword or a list of keywords as in the example above. The complete list of allowable `Type` keywords for 0-tensors is `Complex`, `Real`, `Imaginary`, `Positive`, `Negative`, `NonPositive`, and `NonNegative`. For higher-rank tensors, the `Type` option can be `Real`, `Imaginary`, or `Complex`. The default is always `Real`. If you include one of the keywords indicating positivity or negativity, you may leave out `Real`, which is assumed.

For higher-rank tensors, you can specify tensor symmetries using the `Symmetries` option. For example, to define a symmetric covariant 2-tensor named `h`, type:

```
In[9]:= DefineTensor[ h, 2, Symmetries -> Symmetric ]
```

Other common values for the `Symmetries` option are `Alternating` (or, equivalently, `Skew`) and `NoSymmetries` (the default).

To associate a tensor with a bundle other than the default tangent bundle, use the `Bundle` option:

```
In[10]:= DefineTensor[ omega, 2, Bundle -> fiber,
                        Type -> Complex,
                        Variance -> {Contravariant,Covariant} ]
```

This specifies that `omega` is a complex 2-tensor on the bundle named `fiber`, which is contravariant in the first index and covariant in the second. If the value of the `Variance` option is a list, as in the example above, the length of the list must be equal to the rank of the tensor, and each entry specifies the variance of the corresponding index position. The `Bundle` option may also be typed as a list of bundle names: this means that the tensor is associated with the direct sum of all the bundles in the list. Any time you insert an index into a tensor that does not belong to one of the bundles with which the tensor is associated, you get 0.

After a tensor has been defined, you can change certain of its options, such as `Type` and `Bundle`, by calling `Declare`. For example, to change the 0-tensor `u` defined above to be a complex-valued function, you could type:

```
In[11]:= Declare[ u, Type -> Complex ]
```

Internally, Ricci represents tensors in the form

```
Tensor[ name, {i,j,...}, {k,l,...} ]
```

where `i, j,...` are the tensor indices, and `k, l,...` are indices resulting from covariant differentiation. In input form, you represent an unindexed tensor just by typing its name. Indices are inserted by typing them in brackets after the tensor name, while differentiated indices go inside a second set of brackets.

A tensor of rank k should be thought of as having k "index slots". Each index slot is associated with a bundle or list of bundles (specified in the `DefineTensor` command); if an inserted index is not associated with one of the bundles for that slot, the result is 0. Once all index slots are full, indices resulting from covariant differentiation can be typed in a second set of brackets. For 0-tensors, all indices are assumed to result from differentiation. For example, suppose `eta` is a 2-tensor and `u` is a 0-tensor (i.e., a scalar function). Table 1 shows the input, internal, and output forms for various tensor expressions involving `eta` and `u`.

The conjugate of a complex tensor without indices is represented in input form by `Conjugate[name]`; you can type indices in brackets following this expression just as for ordinary tensors, as in `Conjugate[name] [L[i],L[j]]`. In internal form, a conjugate tensor looks just as in Table 1, except that the name is replaced by the special form `name[Bar]`. In output form, the conjugate of a tensor is represented by a bar over the name.

Indices in any slot can be upper or lower. A metric with lower indices represents the components of the metric on the bundle itself, and a metric with upper indices represents the components of the inverse matrix, which can be interpreted invariantly as the components of the metric on the dual bundle. For any other tensor, if an index is inserted that is not at the "natural altitude" for that slot (based on the `Variance` option when the tensor was defined), it is assumed to have been raised or lowered by means of the metric. For example, if `a` is a covariant 1-tensor, then the components of `a` ordinarily have lower indices. A component with an upper index is interpreted as follows:

INPUT	INTERNAL	OUTPUT
`eta`	`Tensor[eta,{},{}]`	$\mathtt{eta}$
`eta [L[i],U[j]]`	`Tensor[eta,{L[i],U[j]},{}]`	$\mathtt{eta}_{\mathtt{i}}{}^{\mathtt{j}}$
`eta [L[i],U[j]] [L[k]]`	`Tensor[eta,{L[i],U[j]},{L[k]}]`	$\mathtt{eta}_{\mathtt{i}}{}^{\mathtt{j}}{}_{\mathtt{;k}}$
`u`	`Tensor[u,{},{}]`	$\mathtt{u}$
`u [L[a],L[b]]`	`Tensor[u,{},{L[a],L[b]}]`	$\mathtt{u}_{\mathtt{;a\ b}}$

Table 1: Input, internal, and output forms for tensors

$$a^{i} = g^{i\,j}\, a_{j}$$

As explained above, components of tensors ordinarily have lower indices in covariant slots, and upper indices in contravariant slots. Ricci always assumes there is a default local basis (or "moving frame") for each bundle you define, and components of tensors are understood to be computed with respect to this basis. The default contravariant basis vectors (i.e., basis vectors for the bundle itself) are referred to as `Basis[L[i]]`, and the default covariant basis vectors (representing the dual basis for the dual bundle) as `Basis[U[i]]`. (You may specify different names to be used in input and output forms by using the `BasisName` and `CoBasisName` options of `DefineBundle`.)

Note that the index conventions for basis vectors are opposite those for components: this is so that the expansion of tensors in terms of basis vectors will be consistent with the summation convention. The Ricci function `BasisExpand` can be used to expand any tensor expression in terms of the default basis. For example:

```
In[13]:= BasisExpand[alpha]

                         k3
Out[13]= alpha    Basis
              k3
```

2.5 Mathematical functions

If u is a scalar function (0-tensor), you may form other scalar functions by applying real- or complex-valued mathematical functions such as `Log` or `Sin` to u. This is indicated in Ricci's input form and output form simply by typing `Log[u]` or `Sin[u]`. You may also define your own mathematical functions and use them in tensor expressions. In order for Ricci to correctly interpret an expression such as `f[u]`, it must be told that `f` is a mathematical function, and what its characteristics are. You do this by calling `DefineMathFunction`. For example, to define `f` as a complex-valued function of one variable, you could type:

```
In[14]:= DefineMathFunction[ f, Type->Complex ]
```

The `Type` option determines how the function behaves with respect to conjugation, exponentiation, and logarithms. The default is `Real`, which means that the function is always assumed to take real values. Other common options are `Type -> {Positive,Real}`, `Type -> {NonNegative,Real}`, and `Type -> Automatic`, which means that `Conjugate[f[x]] = f[Conjugate[x]]`.

Ricci automatically calls `DefineMathFunction` for the Mathematica functions `Log`, `Sin`, `Cos`, `Tan`, `Sec`, `Csc`, and `Cot`. If you wish to use any other built-in mathematical functions in tensor expressions, you must call `DefineMathFunction` yourself.

2.6 Basic tensor expressions

Tensor expressions are built up from tensors and constants, using the arithmetic operations `Plus`, `Times`, and `Power`, the Mathematica operators `Conjugate`, `Re`, and `Im`, scalar mathematical functions, and the Ricci product, contraction, symmetrization, and differentiation operators described below.

Tensor expressions can be generally divided into three classes: *pure tensor expressions*, without any indices, in which tensor names are combined using arithmetic operations and tensor operators such as `TensorProduct` and `Extd`; *component expressions*, in which all index slots are filled, derivatives are indicated by components of covariant derivatives, and tensor components are combined using only arithmetic operations; and *mixed tensor expressions*, in which some but not all index slots are filled, and which usually arise when expressions of the preceding two types are combined. Component expressions are considered to have rank 0 when they are acted upon by Ricci's tensor operators. In most cases, Ricci handles all three kinds of expressions in a uniform way, but there are some operations, such as computing components of covariant derivatives, that require component expressions.

You may insert indices into any pure or mixed tensor expression, thereby converting it to a component expression, by typing the indices in brackets after the expression, as in `x[L[i],U[j]]`, which is a shorthand input form for the explicit Ricci function `InsertIndices[ x, {L[i],L[j]} ]`. For most tensor expressions, the number of indices inserted must be equal to the rank of the tensor expression. (The only exception is product tensors, described in Section 3 below.) For example, if `alpha` and `beta` are 1-tensors, their tensor product, indicated by `TensorProduct[alpha,beta]`, is a 2-tensor, so you can insert two indices:

```
In[12]:= TensorProduct[alpha,beta] [L[i],L[j]]

Out[12]= alpha   beta
              i      j
```

If `x` is a scalar (rank 0) expression, it can be converted to a component expression by inserting zero indices, as in `InsertIndices[x,{}]` or `x[]`, or by typing `BasisExpand[x]`. For example, `Inner[alpha,beta]` represents the inner product of `alpha` and `beta`:

```
In[13]:= Inner[alpha,beta]//BasisExpand

              k1
Out[13]= alpha    beta
                      k1
```

If you type indices in brackets after a component expression (one in which all index slots are already filled), you get components of covariant derivatives. Thus:

```
In[14]:= %13 [L[j]]

               k1                      k1
Out[14]= alpha       beta     + alpha   beta
                  ;j     k1                   k1 ;j
```

If you type indices in brackets after a scalar (rank-0) expression that is not a component expression (i.e., has unfilled index slots), it is first converted to a component expression before covariant derivatives are taken.

The Ricci functions that can be used to construct tensor expressions are described in later chapters.

2.7 Saving your work

Ricci provides a function called `RicciSave` for saving all the definitions you have made during your current Mathematica session. The command

```
RicciSave["filename"]
```

writes your definitions in the file called `filename` in Mathematica input form. `RicciSave` also writes the current definitions of Ricci's predefined tensors `Basis`, `Curv`, `Tor`, `Conn`, and `Kronecker` (in case you have defined any relations involving them), and of its global variables (those whose names start with `$`). Any previous contents of `filename` are erased. You can read the definitions back in by typing `<<filename`.

3 Products, Contractions, and Symmetrizations

This chapter describes the most important Ricci functions for specifying various kinds of products, contractions, and symmetrizations of tensor expressions. Although Ricci performs some automatic simplification on most of these functions, such as expanding linear combinations of arguments, for the most part these functions are simply maintained unevaluated until you insert indices.

3.1 TensorProduct

The basic product operation for tensors is `TensorProduct`. In input form, this can be abbreviated `TProd`.

```
In[15]:= TProd[alpha, beta]

Out[15]= alpha (X) beta
```

As you can see here, Ricci's output form for tensor products uses an approximation of the tensor product symbol "$\otimes$". Ricci automatically expands tensor products of sums and scalar multiples.

3.2 Wedge

Wedge products (exterior products) of differential forms are indicated by the Ricci operator `Wedge`. In output form, wedge products are indicated by a caret:

```
In[16]:= Wedge[alpha, beta]

Out[16]= alpha ^ beta
```

The arguments to `Wedge` must be alternating tensors. Ricci automatically expands wedge products of sums and scalar multiples, and arranges the factors in lexical order, inserting signs as appropriate.

There are two common conventions for relating wedge products to tensor products. Ricci supports both conventions; which one is in use at any given time is controlled by the global variable `$WedgeConvention`. You should pick one convention and set `$WedgeConvention` at the beginning of your calculation; if you change it, you may get inconsistent results.

In Ricci, the two wedge product conventions are referred to as `Det` and `Alt`. Under the `Alt` convention (the default), the wedge product is defined by

$$\alpha \wedge \beta = \mathrm{Alt}(\alpha \otimes \beta),$$

where Alt is the projection onto the alternating part of the tensor. This convention is used, for example, in the differential geometry texts by Bishop/Goldberg, Kobayashi/Nomizu, and Helgason. Under this convention, on an n-dimensional bundle, the wedge product of all n basis elements is $1/n!$ times the determinant function. Thus, if $e^1, \ldots, e^n$ are basis covectors,

$$e^1 \wedge \ldots \wedge e^n = \frac{1}{n!} \det .$$

The other wedge product convention, called `Det`, is defined by

$$\alpha \wedge \beta = \frac{(p+q)!}{p!q!} \mathrm{Alt}(\alpha \otimes \beta).$$

where α is a p-form and β a q-form. This convention is used, for example, in the texts by Spivak, Boothby, and Warner. The name `Det` comes from the fact that, under this convention, if ω^i are 1-forms and X_j are vectors,

$$\omega^1 \wedge \ldots \wedge \omega^k (X_1, \ldots, X_k) = \det(\omega^i(X_j)).$$

In particular,

$$e^1 \wedge \ldots \wedge e^n = \det .$$

3.3 SymmetricProduct

In Ricci, symmetric products are represented by ordinary multiplication. Thus, if `a` and `b` are 1-tensors, then `a b` represents their symmetric product $a * b$, a 2-tensor. Similarly, `a^2` represents $a * a$.

In input form, you can type a symmetric product just as you would an ordinary product. If you prefer, for clarity, you may indicate symmetric products explicitly using the `SymmetricProduct` function:

```
In[17]:= SymmetricProduct[alpha,beta]

Out[17]= alpha * beta
```

In output form, Ricci inserts explicit asterisks in products of tensors of rank greater than 0, to remind you that multiplication is being interpreted as a symmetric product.

The symmetric product is defined by

$$a * b = \mathrm{Sym}(a \otimes b),$$

where Sym is the projection onto the symmetric part of a tensor.

3.4 Dot

`Dot` is a Mathematica function that is given an additional interpretation by Ricci. If `a` and `b` are tensor expressions, then `Dot[a,b]` or `a.b` is the tensor formed by contracting the last index of `a` with the first index of `b`. It is easiest to see what this means by inserting indices. For example, suppose `a` is a 2-tensor and `e` is a 3-tensor:

```
In[18]:= (a.e) [L[i],L[j],L[k]]


                   k4
Out[18]= a        e
          i k4       j k
```

`Dot` can be applied to three or more tensor expressions, as long as all of the tensors except possibly the first and last have rank two or more. Ricci expands dot products of sums and scalar multiples automatically. Dot products of 1-tensors are automatically converted to inner products.

3.5 Inner

The Mathematica function `Inner` is given an additional interpretation by Ricci. If `a` and `b` are tensor expressions of the same rank, `Inner[a,b]` is their inner product, taken with respect to the default metric(s) on the bundle(s) with which `a` and `b` are associated. In output form, `Inner[a,b]` appears as `<a, b>`. For example:

```
In[19]:= Inner[a,b]

Out[19]= <a, b>

In[20]:= % // BasisExpand

          k5 k6
Out[20]= a       b
                  k5 k6
```

3.6 HodgeInner

`HodgeInner` is the inner product ordinarily used for differential forms. This differs from the usual inner product by a constant factor, determined by the global variable `$WedgeConvention`. It is defined in such a way that, if e^i are orthonormal 1-forms, then the k-forms

$$e^{i_1} \wedge \ldots \wedge e^{i_k}, \quad i_1 < \ldots < i_k$$

are orthonormal. In output form, `HodgeInner[alpha,beta]` appears as `<<a, b>>`.

3.7 Int

If `v` is a vector field and `alpha` is a differential form, `Int[v,alpha]` represents interior multiplication of `v` into `alpha` (usually denoted $i_v\alpha$ or $v \lrcorner \alpha$), with a normalization factor depending on `$WedgeConvention`, chosen so that `Int[ v, _ ]` is an antiderivation.

More generally, if `alpha` and `beta` are any alternating tensor expressions with `Rank[alpha]` $\leq$ `Rank[beta]`, then `Int[alpha,beta]` represents the generalized interior product of `alpha` into `beta` (sometimes denoted $\alpha \vee \beta$). `Int[ alpha, _ ]` is defined as the adjoint (with respect to the Hodge inner product) of wedging with `alpha` on the left: if `alpha`, `beta`, and `gamma` are alternating tensors of ranks a, b, and $a + b$, respectively, then

```
<<alpha ^ beta, gamma>> = <<beta, Int     [gamma]>>.
                                     alpha
```

3.8 Alt

If `x` is any tensor expression, `Alt[x]` is the alternating part of `x`. If `x` is already alternating, then `Alt[x] = x`. Note that `Alt` expects its argument to be a pure or mixed tensor expression. This means that you must apply `Alt` *before* you insert indices. If `Alt` is applied to a component expression, it does nothing, because a component expression is considered to have rank 0.

3.9 Sym

If `x` is any tensor expression, `Sym[x]` is the symmetric part of `x`. If `x` is already symmetric, then `Sym[x] = x`. Note that `Sym` expects its argument to be a pure or mixed tensor expression. This means that you must apply `Sym` *before* you insert indices. If `Sym` is applied to a component expression, it does nothing, because a component expression is considered to have rank 0.

4 Derivatives

This chapter describes the differentiation operators provided by Ricci. The operators `Del`, `Div`, `Grad`, `Extd`, `ExtdStar`, and `Lie` below are Ricci "primitives"; after some simplification, they are maintained unevaluated until you insert indices. The rest of the operators described here are immediately transformed into expressions involving the primitive operators.

4.1 Del

The basic differentiation operator in Ricci is `Del`. If `x` is a tensor expression, `Del[x]` represents the total covariant derivative of `x`. (`Del` stands for the mathematical symbol "∇".) If `x` is a tensor of rank k, `Del[x]` is a tensor of rank $k+1$. If `x` is a section of bundle b, `Del[x]` is a section of the tensor product of b with its underlying (co)tangent bundle. The extra index slot resulting from differentiation is always the last one, and is always covariant.

When you insert indices into `Del[x]`, you get components of the covariant derivative of `x`. For example, suppose `a` is a 2-tensor:

```
In[21]:= Del[a]

Out[21]= Del[a]

In[22]:= % [L[i],L[j],L[k]]

Out[22]= a
          i j ;k
```

The operator `Del` can also be used to take the covariant derivative of a tensor with respect to a specific vector. If `v` is a 1-tensor expression, `Del[v,x]` represents the covariant derivative of `x` in the direction `v`; this is the contraction of `v` with the last index of `Del[x]`:

```
In[23]:= Del[v,a]

Out[23]= Del   [a]
            v

In[24]:= % [L[i],L[j]]

                    k5
Out[24]= a         v
          i j ;k5
```

Here `v` can be a contravariant or covariant 1-tensor; if it is covariant, it is converted to a vector field by means of the metric. If `u` is a scalar function (i.e., a 0-tensor), then `Del[v,u]` is just the ordinary directional derivative of `u` in the direction `v`.

If `v` is a contravariant basis vector such as `Basis[L[i]]`, Ricci uses a special shorthand output form for `Del[v,a]`:

```
In[25]:= Del[ Basis[L[i]], a ]

Out[25]= Del   [a]
            i
```

WARNING: Some authors use a notation such as $\nabla_i a_{j\,k}$ to refer to the (j, k, i)-component of the total covariant derivative ∇a. This is *not* consistent with Ricci's convention; to Ricci, the expression

```
Del  [a    ]
   i   j k
```

is the directional derivative of the *component* $a_{j\,k}$ in the i-direction. It is not the component of a tensor; it differs from the component $a_{j\,k;i}$ by terms involving the connection coefficients.

4.2 CovD

If `x` is a component expression, the `L[i]` component of the covariant derivative of `x` is represented in input form either by `CovD[x,{L[i]}]` or by the abbreviated form `x[L[i]]`. Multiple covariant derivatives are indicated by `CovD[x,{L[i],L[j],...}]` or `x[L[i],L[j],...]`. If `x` is a scalar expression with some unfilled index slots, Ricci automatically inserts indices to convert it to a component expression before taking covariant derivatives. Ricci automatically expands covariant derivatives of sums, products, and powers. For example:

```
In[26]:= (1 + a[L[i]] b[U[i]]) ^ 2

                   i  2
Out[26]= (1 + a   b   )
               i

In[27]:= % [L[j]]

                    i         i                  k6
Out[27]= 2 (a      b    + a   b      ) (1 + a   b   )
             i ;j          i      ;j          k6
```

4.3 Div

`Div[x]` represents the divergence of the tensor expression `x`. This is just the covariant derivative of `x` contracted on its last two indices. If `x` is a k-tensor, `Div[x]` is a $(k-1)$-tensor. Ricci assumes that the last index of `x` is associated with the underlying tangent bundle of `x`'s bundle.

For example, suppose `a` is a 2-tensor:

```
In[28]:= Div[a] [L[i]]

                k8
Out[28]= a
          i k8 ;
```

`Div` is the formal adjoint of `-Del`.

4.4 Grad

If `x` is any tensor expression, `Grad[x]` is the gradient of `x`. It is the same as `Del[x]`, except that the last index is converted to contravariant by means of the metric. If `x` is a scalar function (0-tensor), then `Grad[x]` is a vector field.

4.5 Laplacian

`Laplacian[x]` is the covariant Laplacian of the tensor expression `x`. There are two common conventions in use for the Laplacian, and both are supported by Ricci. Which convention is in effect is controlled by the value of the global variable `$LaplacianConvention`. When `$LaplacianConvention = DivGrad` (the default), `Laplacian[x]` is automatically replaced by `Div[Grad[x]]`. If `$LaplacianConvention` is set to `PositiveSpectrum`, then `Laplacian[x]` is replaced by `-Div[Grad[x]]`.

4.6 Extd

`Extd[x]` represents the exterior derivative of the differential form `x`. In output form, `Extd[x]` appears as `d[x]`. Ricci automatically expands exterior derivatives of sums, scalar multiples, powers, and wedge products.

WARNING: It is tempting to type `d[x]` in input form. Because the symbol `d` has not been defined as a Ricci function, Ricci simply returns `d[x]` unevaluated. You may not be able to tell immediately from Ricci's output that it is not interpreting `d[x]` as exterior differentiation, unless you happen to notice that it has not expanded derivatives of sums or products that appear in the argument to `d`.

4.7 ExtdStar

`ExtdStar` is the formal adjoint of `Extd` with respect to the Hodge inner product. It is used in constructing the Laplace-Beltrami operator, described in Section 4.8 below. If `x` is a differential k-form, `ExtdStar[x]` is a $(k-1)$-form. In output form, `ExtdStar` appears as shown here:

```
In[29]:= ExtdStar[alpha]

             *
Out[29]= d  [alpha]
```

4.8 LaplaceBeltrami

`LaplaceBeltrami[x]` is the Laplace-Beltrami operator $\Delta = dd^* + d^* d$ applied to `x`, which is assumed to be a differential form. It is automatically replaced by `Extd[ExtdStar[x]] + ExtdStar[Extd[x]]`.

4.9 Lie

If `v` is a vector field (i.e., a contravariant 1-tensor) and `x` is any tensor expression, `Lie[v,x]` represents the Lie derivative of `x` with respect to `v`. When `v` and `x` are both vector fields, `Lie[v,x]` is their Lie bracket, and Ricci puts `v` and `x` in lexical order using the skew-symmetry of the Lie bracket.

Lie derivatives of differential forms are not automatically expanded in terms of exterior derivatives and interior multiplication, because there may be some situations in which you do not want this transformation to take place. You can cause them to be expanded by applying the rule `LieRule`:

```
In[13]:= Lie[v,beta]

Out[13]= Lie  [beta]
            v

In[14]:= % /. LieRule

Out[14]= d[Int  [beta]] + Int  [d[beta]]
              v               v
```

5 Simplifying and Transforming Expressions

This chapter describes commands that can be used to simplify tensor expressions, or to perform various transformations such as expanding covariant derivatives in terms of ordinary directional derivatives or *vice versa*.

The commands in this chapter that take only one argument can be applied only to selected terms of an expression. For example, `TensorSimplify[x,n]` simplifies only term `n`, leaving the rest of the expression unchanged. `TensorSimplify[x,{n1,n2,n3,...}]` simplifies terms `n1`, `n2`, `n3`, ..., combining terms if possible, but leaving the rest of `x` unchanged. The Reference List, Chapter 8, describes which commands accept this syntax.

5.1 TensorSimplify

The most general simplification command provided by Ricci is `TensorSimplify`. `TensorSimplify[x]` attempts to put an indexed tensor expression `x` into a canonical form. The rules applied by TensorSimplify may not always result in the simplest-looking expression, but two component expressions that are mathematically equal will usually be identical after applying `TensorSimplify`.

`TensorSimplify` expands products and powers of tensors, uses metrics to raise and lower indices, tries to rename all dummy indices in a canonical order, and collects all terms containing the same tensor factors but different constant factors. When there are two or more dummy index pairs in a single term, `TensorSimplify` tries exchanging names of dummy indices in pairs, and chooses the lexically smallest expression that results. This algorithm may not always recognize terms that are equal after more complicated rearranging of dummy index names; an alternative command, `SuperSimplify`, works harder to get a canonical expression, at the cost of much slower execution.

`TensorSimplify` makes only those simplificatons that arise from the routine changing of dummy index names and the application of symmetries and algebraic rules that the user could easily check. More complicated simplifications, such as those that arise from the Bianchi identities or from commuting covariant derivatives, are done only when you request them. The basic reason for this distinction is so that you will not be confronted by a mystifyingly drastic and unxplainable simplification. To reorder covariant derivatives, that is, the ones that come after ";", use `OrderCovD` or `CovDSimplify`. To apply Bianchi identities, apply one of the rules `BianchiRules`, `FirstBianchiRule`, `SecondBianchiRule`, or `ContractedBianchiRules`.

`TensorSimplify` calls `CorrectAllVariances`, `TensorExpand`, `AbsorbMetrics`, `PowerSimplify`, `RenameDummy`, `OrderDummy`, and `CollectConstants`. Any of these commands can be used individually to provide more control over the simplification process.

5.2 SuperSimplify

`SuperSimplify[x]` does the same job as `TensorSimplify[x]`, but works harder at renaming dummy index pairs to find the lexically smallest version of the expression. If there are two or more dummy index pairs in any term of `x` that refer to the same bundle, `SuperSimplify` tries renaming the dummy indices in all possible permutations, and chooses the lexically smallest result. If there are k dummy index pairs per term, the time taken by `SuperSimplify` is proportional to $k!$, while the time taken by `TensorSimplify` is proportional to k^2. Thus `SuperSimplify` can be *very* slow, especially when there are more than about 4 dummy pairs. This command should be used sparingly, only when you suspect that some terms are equal but `TensorSimplify` has not made them look the same.

5.3 TensorExpand

`TensorExpand[x]` expands products and positive integral powers in `x`, just as `Expand` does, but maintains correct dummy index conventions and does not expand constant factors.

`TensorExpand` is called automatically by `TensorSimplify`.

5.4 AbsorbMetrics

`AbsorbMetrics[x]` causes any metric components that appear contracted with other tensors in `x` to be used to raise or lower indices; the metric components themselves are eliminated from the expression. For example:

```
                 j k
Out[29]= a      g
          i j

In[29]:= AbsorbMetrics[%]

             k
Out[30]= a
          i
```

`AbsorbMetrics` is called automatically by `TensorSimplify`.

5.5 RenameDummy

`RenameDummy[x]` changes the names of dummy indices in `x` to standard names, choosing index names in alphabetical order from the list associated with the appropriate bundle, and skipping those names that already appear in `x` as free indices. When the list of index names is exhausted, computer-generated names of the form `k`n are used, where `k` is the last index name in the list and n is an integer. `RenameDummy` does not try to make the "best" choice of index names for the expression; use `OrderDummy` to do that.

`RenameDummy` is called automatically by `TensorSimplify`.

5.6 OrderDummy

`OrderDummy[x]` attempts to put the dummy indices occurring in the tensor expression `x` in a "canonical form". All pairs of dummy indices are ordered so that the lower member appears first whenever possible. Then `OrderDummy` tries various permutations of the dummy index names in each term of `x`, searching for the lexically smallest version of the expression among all equivalent versions. `OrderDummy` has an option called `Method`, which controls how hard it works to find the best arrangement of dummy index names. See the Reference List, Chapter 8, for details.

`OrderDummy` is called automatically by `TensorSimplify`.

5.7 CollectConstants

`CollectConstants[x]` groups together terms in the tensor expression `x` having the same tensor factors but different constant factors, and performs some simplification on the constant factors.

`CollectConstants` is called automatically by `TensorSimplify`.

5.8 FactorConstants

`FactorConstants[x]` applies the Mathematica function `Factor` to the constant factor in each term of `x`.

5.9 SimplifyConstants

`SimplifyConstants[x]` applies the Mathematica function `Simplify` to the constant factor in each term of `x`.

5.10 BasisExpand

If `x` is any tensor expression, `BasisExpand[x]` expands all tensors in `x` into component expressions multiplied by the default basis vectors and covectors. For example, suppose `a` is a 2-tensor:

```
In[34]:= BasisExpand[2 a]

                             k1           k2
Out[34]= 2 a        Basis   (X) Basis
            k1 k2
```

Because of the limitations of the index notation, there is no provision for expanding tensors automatically with respect to any basis other than the default one; you can usually accomplish the same thing by using `DefineRule` to create transformation rules between the default basis and other bases.

If `x` is a scalar (rank 0) expression, `BasisExpand[x]` causes all index slots to be filled, thus converting `x` to a component expression.

5.11 BasisGather

Sometimes you may want to convert a component expression for a tensor back to pure tensor form. `BasisGather` allows you to do this. For example, we can undo the effect of `BasisExpand` in `Out[34]` above:

```
In[35]:= BasisGather[%34,a]

Out[35]= 2 a
```

The second argument to `BasisGather` can in principle be any tensor expression, although `BasisGather` may not recognize the basis expansion of complicated expressions. It works best if you apply `TensorExpand` (or `TensorSimplify`) before applying `BasisGather`. You can gather several tensor expressions at once by putting a list of expressions in the second argument.

5.12 CovDExpand

`CovDExpand[x]` expands all covariant derivatives in `x` into ordinary directional derivatives and connection components. Directional derivatives of components of tensors are represented by expressions like `Del[ Basis[L[i]], a[L[j],L[k]] ]`. For example:

```
Out[35]= a
          j k ;i

In[36]:= CovDExpand[%]

                                        k3                    k4
Out[36]= Del   [a    ] - a       Conn          - a       Conn
            i    j k       k3 k       j     i       j k4       k    i
```

Here `Conn` represents the coefficients of the default connection with respect to the default basis. See Section 7.5, "Connections, torsion, and curvature", for more information.

5.13 ProductExpand

`ProductExpand[x]` expands symmetric products and wedge products of 1-tensors that occur in `x`, and rewrites them in terms of tensor products. For example, assuming `$WedgeConvention = Alt`, you would get the following result:

```
              i         j
Out[5]= Basis    ^ Basis

In[6]:= %//ProductExpand

         1      i           j    1      j           i
Out[6]= - Basis    (X) Basis    - - Basis    (X) Basis
         2                       2
```

You can reverse the effect of `ProductExpand` by applying `Alt` to an alternating tensor expression, or `Sym` to a symmetric expression; these operators replace tensor products by wedge or symmetric products.

5.14 PowerSimplify

`PowerSimplify[x]` performs various simplifications on powers that appear in `x`, such as transforming `a^p b^p` to `(a b)^p`, and `(a^b)^c` to `a^(b c)` when possible, and expanding and collecting constants in expressions that appear as base or exponent of a power.

`PowerSimplify` is called automatically by `TensorSimplify`.

5.15 CorrectAllVariances

`CorrectAllVariances[x]` changes the variances (upper to lower or lower to upper) of all indices occurring in `x` whose variances are not correct for their positions, by inserting appropriate metric coefficients. For example, suppose `v` is a vector field (i.e., a contravariant 1-tensor):

```
                j
Out[41]= v
          i ;  k

In[42]:= CorrectAllVariances[%]

                 j k4   k3
Out[42]= g      g       v
          i k3              ;k4 k
```

5.16 NewDummy

`NewDummy[x]` replaces all dummy index pairs in `x` with computer-generated dummy index names of the form `k`n, where `k` is the last name in the list of indices associated with the appropriate bundle, and n is a unique integer. This is sometimes useful when you want to insert an index with the same name as a dummy name that appears in `x`. For example, suppose `a` is a 2-tensor. After applying `TensorSimplify`, the output from `BasisExpand[a]` becomes:

```
                     i           j
Out[31]= a     Basis   (X) Basis
          i j
```

If you want the (i, k)-component of this expression, you cannot just type `%[L[i],L[k]]`, because `i` would then appear twice as a lower index—once in the output expression `Out[31]`, and once in the inserted index. The correct procedure is to use `NewDummy`:

```
In[32]:= NewDummy[%] [L[i],L[k]]

Out[33]= a
          i k
```

5.17 CommuteCovD

`CommuteCovD[ x, L[i], L[j] ]` changes all adjacent occurrences in the tensor expression `x` of indices `L[i],L[j]` (in the given order) after the ";" to `L[j],L[i]` by adding appropriate curvature and torsion terms. For example, suppose `a` is a 2-tensor on a Riemannian tangent bundle (assuming `$RiemannConvention = SecondUp`):

```
Out[35]= a
          i j ;k l

In[36]:= CommuteCovD[ %, L[k], L[l] ]

                                l1                  l2
Out[36]= a           + a      Rm           + a      Rm
          i j ;l k     l1 j    i    k l     i l2    j    k l
```

5.18 OrderCovD

`OrderCovD[x]` puts all derivative indices (those appearing after ";") in the tensor expression `x` into lexical order, by adding appropriate curvature and torsion terms. `OrderCovD` is used by `CovDSimplify`.

5.19 CovDSimplify

`CovDSimplify[x]` attempts to simplify `x` as much as possible, putting all dummy indices in lexical order, including those that result from differentiation. `CovDSimplify` calls `TensorSimplify`, then `OrderCovD`, then `TensorSimplify` again. For very complicated expressions, it is probably more efficient to use the other simplification commands individually.

5.20 LowerAllIndices

`LowerAllIndices[x]` converts all upper indices in `x` (except those appearing on `Basis` covectors or metrics) to lower indices by inserting metrics as needed.

5.21 TensorCancel

`TensorCancel[x]` attempts to simplify products and quotients of tensor expressions by combining and canceling factors that are equal even though they have different dummy index names. For example:

```
                     i  3                j  -1
Out[37]= (1 + u    u   )  (1 + u    u   )
               ;i    ;              ;j    ;

In[38]:= TensorCancel[%]

                     i  2
Out[38]= (1 + u    u   )
               ;i    ;
```

6 Defining Relations Between Tensors

Because the internal and external forms of tensors differ, you should never use the name of a tensor on the left-hand side of an assignment or rule, such as `a=b`, `a:=b`, `a->b`, or `a:>b`. Ricci provides two commands, `DefineRelation` and `DefineRule`, to create such assignments and rules. This is a common source of error. To emphasize:

NEVER USE THE NAME OF A TENSOR ON THE LEFT-HAND SIDE OF AN ASSIGNMENT OR RULE; USE `DefineRelation` OR `DefineRule` INSTEAD.

6.1 DefineRelation

If you want to cause one tensor always to be replaced by some other tensor expression, or to be replaced only when certain indices are inserted, you can use `DefineRelation`. The basic syntax is shown by the following example:

```
In[39]:= DefineRelation[ a, 2 b ]

Relation defined.

In[40]:= a[L[i],L[j]]

Out[40]= 2 b
             i j
```

This use of `DefineRelation` specifies that every occurrence of the tensor `a` should be replaced by `2 b`. The replacement specified in such a call to `DefineRelation` is made whether or not indices are inserted into `a`.

The first argument to `DefineRelation` must be a single tensor, with or without indices; it cannot be a more complicated expression. (The principal reason is that Mathematica requires every relation to be associated with a specific symbol, and Ricci's philosophy is to associate your definitions only with your own symbols, so that they can all be saved by Ricci. If you tried to define a relation involving a complicated tensor expression, it would have to be associated with a Ricci or system function.) You can use `DefineRule`, described below, to define transformation rules whose left-hand sides are arbitrary tensor expressions.

You can cancel any relation created by `DefineRelation` by calling `UndefineRelation`. The single argument to `UndefineRelation` must be exactly the same as the first argument in the corresponding call to `DefineRelation`. For example:

```
In[41]:= UndefineRelation[a]

Relation undefined.
```

You can also use `DefineRelation` to replace a tensor only when it appears with certain indices. For example, suppose `e` is a 3-tensor:

```
In[42]:= DefineRelation[ e[L[i],U[i],L[j]], v[L[j]] ]

Relation defined.
```

This tells Ricci that the indexed version of `e` is to be replaced by `v` when the first two indices of `e` are contracted with each other. (Note that you do *not* use underscores in the dummy index names for `DefineRelation`, as you would in defining a relation directly with Mathematica.) The replacement takes place regardless of the actual names of the indices or whether they are upper or lower, provided the corresponding indices refer to the same bundles as `i` and `j`. For example:

```
In[43]:= e[U[j],L[j],L[k]]

Out[43]= v
          k
```

In general, the relation created by `DefineRelation` recognizes any equivalent relations resulting from raising and lowering indices, changing index names, conjugation, and inserting extra indices, provided that all indices refer to the same bundles as the corresponding indices that were used in the call to `DefineRelation`. It does not recognize relations resulting from symmetries, however—you must define all such equivalent relations explicitly.

You can specify that the relation should be applied only under certain conditions by giving `DefineRelation` a third argument, which is a True/False condition that must be satisfied before the relation is applied. The condition may refer to the index names used in the first argument. For example, suppose `u` is a scalar function on a Riemannian tangent bundle, and you would like the first and second covariant derivatives of `u` always to be placed in lexical order (this is not done automatically by Ricci). You could define the following conditional relation:

```
In[44]:= DefineRelation[u[L[i],L[j]], u[L[j],L[i]],
           !IndexOrderedQ[{L[i],L[j]}]]

Relation defined.

In[45]:= u[L[j],L[i]]

Out[45]= u
          ;i j
```

6.2 DefineRule

Sometimes you may want to define a transformation rule that is not applied automatically, but only when you request it. Such transformations are done in Mathematica via rules. The Ricci function `DefineRule` allows you to define rules whose left-hand sides involve tensors.

For example, here's how to define rules that replace the tensor `a` by `2 b` and *vice versa*:

```
In[52]:= DefineRule[ atob, a, 2 b]

Rule defined.

In[53]:= DefineRule[ btoa, b, a/2]

Rule defined.

In[54]:= a[L[i],L[j]]

Out[54]= a
          i j

In[55]:= % /. atob

Out[55]= 2 b
            i j
```

```
In[56]:= % /. btoa

Out[56]= a
          i j
```

`DefineRule` has another advantage over `DefineRelation`: the left-hand side of the transformation rule defined by `DefineRule` can be any tensor expression, not just a single tensor with or without indices. Thus, suppose you want to convert the inner product of `a` and `b` to 0 wherever it occurs. You could define the following rules:

```
In[57]:= DefineRule[ ortho, Inner[a,b], 0]

Rule defined.

In[58]:= DefineRule[ ortho, a[L[i],L[j]] b[U[i],U[j]], 0]

Rule defined.
```

The symbol `ortho` is now a list containing these two transformation rules. The first rule transforms the pure-tensor expression for `<a, b>` to 0, while the second rule takes care of the component expression. Here's how it works:

```
In[60]:= a[L[j],L[k]] b[U[j],U[k]]

                j k
Out[60]= a     b
          j k

In[61]:= % /. ortho

Out[61]= 0
```

Each time you call `DefineRule`, the new transformation rule you give is appended to a list containing ones already defined under the same name. To start over from scratch, you can include the option `NewRule -> True` (or just execute the assignment `ortho=.`).

The rule created by `DefineRule` recognizes any equivalent substitutions resulting from raising and lowering indices and changing index names. In addition, if the second argument is a single tensor name with or without indices, it recognizes equivalent substitutions resulting from conjugation and inserting extra indices.

7 Special Features

7.1 One-dimensional bundles

One-dimensional bundles receive special treatment in Ricci. Since there is only one basis element, Ricci uses only one index name for all indices referring to the bundle. This is the only case in which an index can legitimately appear more than once as a lower or upper index in the same term.

When you define a one-dimensional bundle, you may only give one index name. When Ricci generates dummy indices for the bundle, it uses only this index name instead of generating unique names.

If you define a one-dimensional bundle with the option `OrthonormalFrame -> True`, the metric coefficient with respect to the default basis is always equal to 1. `TensorSimplify` then puts all indices associated with this bundle at their natural altitude, as specified in the corresponding call to `DefineTensor`.

7.2 Riemannian metrics

You specify that a bundle is a tangent bundle with a Riemannian metric by including the option `MetricType -> Riemannian` in your call to `DefineBundle`. This automatically causes the torsion of the default connection to be set to 0, and defines the Riemannian, Ricci, and scalar curvature tensors. By default, these are named `Rm`, `Rc`, and `Sc`; you may give them any names you wish by including the `DefineBundle` options `RiemannTensor -> name`, `RicciTensor -> name`, and `ScalarCurv -> name`. If you define more than one Riemannian tangent bundle, you must give different names to the curvatures on different bundles.

On a Riemannian tangent bundle, curvature components are automatically converted to components of the Riemannian curvature tensor, and contractions of the curvature tensor are automatically converted to the Ricci or scalar curvature as appropriate. The first and second Bianchi identities, however, are not automatically applied; you can specifically apply them using the Ricci rules `FirstBianchiRule`, `SecondBianchiRule`, `ContractedBianchiRules`, and `BianchiRules`, described in the Reference List, Chapter 8.

There are two common sign conventions in use for indices of the Riemannian curvature tensor. Ricci supports both; you can control which one is used for any particular Riemannian tangent bundle either by setting the global variable `$RiemannConvention` or by including the `RiemannConvention` option in the call to `DefineBundle`. See `DefineBundle` in the Reference List, Chapter 8, for details.

You may convert components of the Riemannian, Ricci, and scalar curvature tensors of a Riemannian tangent bundle to connection components (Christoffel symbols) and their derivatives by applying the Ricci rule `CurvToConnRule`. Connection components, in turn, can be converted to expressions involving directional derivatives of the metric components by applying `ConnToMetricRule`. (Note, however, that the latter rule applies only to bundles that have been given the option `CommutingFrame -> True`, since this is the only case in which the connection components can be expressed purely in terms of derivatives of the metric.)

7.3 Matrices and 2-tensors

Because every bundle has a default metric, any 2-tensor on a given bundle can be interpreted as a linear endomorphism of the bundle. Ricci supports various matrix computations with such tensors.

The basic operation is matrix multiplication, which is indicated by `Dot`. For example, if `a` and `b` are 2-tensors, then `a.b` or `Dot[a,b]` represents their matrix product. If `v` is a 1-tensor, `a.v` represents the matrix `a` acting on the vector `v`. Because the metric is used automatically to raise and lower indices, no distinction is made between row and column vectors, or between covariant and contravariant tensors, for this purpose.

The identity matrix on any bundle can be represented by the tensor `Kronecker`, whose components are the Kronecker delta symbols. Ricci automatically recognizes some basic relations involving multiplication by `Kronecker`, such as `a.Kronecker = Kronecker.a = a`.

The inverse of any 2-tensor expression `a` is represented by `Inverse[a]`. When you insert indices into an inverse tensor, Ricci in effect creates a new tensor whose name is `Inverse[a]`. For example:

```
In[66]:= Inverse[a]

          -1
Out[66]= a

In[67]:= % [U[i],U[j]]

            -1 i j
Out[67]= (a     )
```

Ricci recognizes that multiplication by the inverse of a tensor yields the identity tensor, whether as components or as pure tensors:

```
In[68]:= a . b . Inverse[b]

Out[68]= a

In[69]:= a[L[i],L[j]] Inverse[a] [U[j],U[k]]

                   k
Out[69]= Kronecker
                  i
```

You can express the trace of a 2-tensor (with respect to the default metric) by `Tr[a]`, its transpose by `Transpose[a]`, and its determinant by `Det[a]`. When you insert indices into `Tr[a]` or `Transpose[a]`, Ricci computes the components explicitly in terms of the components of `a`. For `Det`, however, there is no straighforward way to express the determinant of a matrix using the index conventions, so `Det[a]` is maintained unevaluated. Ricci's differentiation commands compute derivatives of `Det[a]` in terms of derivatives of components of `a` and `Inverse[a]`.

7.4 Product tensors

It often happens that a tensor is associated with the tensor product of two or more bundles. In Ricci, such a tensor is called a *product tensor*. A product tensor is defined by giving a list of positive integers as the rank of the tensor in the `DefineTensor` command. The rank of the tensor is the sum of all the numbers in the list; each number in the list corresponds to a group of indices, which can have its own symmetries and can be associated to a separate bundle or direct sum of bundles. Indices can be inserted into a product tensor "one group at a time": for example, if `t` is a product tensor whose rank is specified as {2,3}, you can insert either two or five indices. Once the first two indices are inserted, the resulting expression is considered to be a 3-tensor, with the symmetries and bundles associated with the second group of indices.

The `Bundle` and `Symmetries` options of `DefineTensor` have special interpretations for product tensors.

- `Bundle -> {bundle1,bundle2,...}`: the list of bundles must be the same length as the list of ranks. Each bundle in the list is associated with the corresponding set of index slots. Other forms are `Bundle -> bundle`, which means that all indices are associated with the same bundle, and `Bundle -> {{b1,b2},{b3,b4},...}`, which means that each set of index slots is associated with a direct sum of bundles.

- `Symmetries -> {sym1,sym2,...}`: again, the list of symmetries must be the same length as the list of ranks. Each symmetry applies to the corresponding set of index slots. The only other acceptable form for product tensors is `Symmetries -> NoSymmetries`, which is the default.

One common use of product tensors is in defining arbitrary connections and their curvatures. In fact, the built-in tensors `Conn` and `Curv` are defined as product tensors.

For example, the following command defines a rank-4 tensor whose first two index slots refer to the sum of `fiber` and its conjugate, and whose last two refer to `tangent`. It is skew-symmetric in the last two indices, with no symmetries in the first two. It is contravariant in the second index and covariant in the other three.

```
In[62]:= DefineTensor[
           omega, {2,2},
           Bundle -> { {fiber,Conjugate[fiber]}, tangent },
           Symmetries -> {NoSymmetries,Skew},
           Variance -> {Co,Con,Co,Co} ]
```

When you insert indices into this tensor, you may either insert the first two indices only, or insert all four at once. (There is no provision for inserting only the last two indices.) When the first two indices are inserted, the result is a skew-symmetric covariant 2-tensor associated with the `tangent` bundle. For example, assuming index names `a`, `b`, `c` are associated with `fiber`:

```
In[5]:= omega[L[a],U[b]]

                 b
Out[5]= omega
               a

In[6]:= %//BasisExpand

                 b              k6          k7
Out[6]= omega            Basis     ^ Basis
               a   k6 k7
```

Another way to use a product tensor is to represent a section of $Hom(a,b)$, where a and b are vector bundles. Suppose you define a tensor `hom` as follows:

```
In[35]:= DefineTensor[hom, {1,1}, Bundle -> {b,a},
                                  Variance -> {Con,Co}]
```

Then `hom` is a section of $b \otimes a^*$, or equivalently a homomorphism from a to b.

7.5 Connections, torsion, and curvature

Ricci assumes that every bundle comes with a default connection that is compatible with its metric. (This assumption is forced by the limitations of the index convention, since the notation for components of covariant derivatives gives no indication what connection is being used.)

The name for the default connection on any bundle is `Conn`. `Conn` is a product tensor of rank {2,1}, whose first two indices can come from any bundle, and whose third index is associated with the underlying tangent bundle of the first two. When you insert the first two indices, you get an entry of the matrix of connection 1-forms corresponding to the default basis, defined by the relation:

$$\nabla Basis_i = Basis_j \otimes Conn_i{}^j.$$

When you insert the third index, you get the connection coefficients (Christoffel symbols) for the default connection with respect to the default basis. These are defined by:

$$\nabla_{Basis_k} Basis_i = Conn_i{}^j{}_k Basis_j.$$

For example, suppose that indices `a`, `b`, `c` refer to the bundle `fiber`, and `i`, `j`, `k` refer to the underlying tangent bundle. The connection 1-forms of `fiber` would be expressed as:

```
                b
Out[15]= Conn
              a

In[16]:= BasisExpand[%]

                b          k8
Out[16]= Conn         Basis
              a   k8
```

Because the default metric on any bundle is assumed to be compatible with its metric, the coefficients of the metric satisfy the compatibility equation

$$d(g_{ij}) = Conn_{ij} + Conn_{ji},$$

where the second index of $Conn$ has been lowered by the metric. This relation is not applied automatically by Ricci, but you can force Ricci to replace derivatives of the metric by connection forms by applying the rule `CompatibilityRule`.

A connection on the tangent bundle of a manifold may or may not have torsion. Ricci assumes that connections are torsion-free, unless you specify that a particular connection may have non-vanishing torsion by adding the option `TorsionFree -> False` to the `DefineBundle` command when the bundle is created. The name of the torsion tensor on any bundle is `Tor`. It is a product tensor of rank {1,2}, whose first index can be associated with any (subbundle of a) tangent bundle, and whose last two are associated with the full underlying tangent bundle (or bundles, if the tangent bundle is a direct sum). It is skew-symmetric in its last two indices. The torsion is defined by

$$\nabla_j Basis_k - \nabla_k Basis_j - [Basis_j, Basis_k] = Tor^i{}_{jk} Basis_i.$$

When the first index is inserted, `Tor[U[i]]` represents the torsion 2-form associated with the basis 1-form `Basis[U[i]]`. The first structure equation relates the exterior derivatives of basis covectors on the tangent bundle to the connection and torsion forms. Ricci does not perform this transformation automatically, but you can cause it to be performed by applying `FirstStructureRule`:

```
                  i
Out[36]= d[Basis  ]

In[37]:= % /. FirstStructureRule

         1    i          k6           i
Out[37]= - Tor   + Basis     ^ Conn
         2                          k6
```

The 1/2 appears because, by Ricci's default wedge product convention and index conventions, the basis expansion of `Tor[U[i]]` has to be

```
   i           j           k        i            j        k
Tor        Basis  (X) Basis   =  Tor        Basis  ^ Basis
     j k                              j k
```

without the factor of 1/2 that appears in the standard notation used in many differential geometry texts. The 1/2 does not appear in the output from `FirstStructureRule` when `$WedgeConvention = Det`, because in that case,

```
   i            j          k     1    i            j        k
Tor       Basis  (X) Basis   =   - Tor       Basis  ^ Basis
     j k                         2      j k
```

The curvature associated with the default connection is called `Curv`. This is a product tensor of rank {2,2}, which is skew-symmetric in both pairs of indices (because the default connection is always assumed to be compatible with the metric). The first two indices of `Curv` can refer to any bundle, and the last two refer to its underlying tangent bundle. When you insert the first two indices, you get the matrix of curvature 2-forms associated with the standard basis. When all four indices are inserted, you get the components of the curvature tensor, defined by

$$\nabla_i \nabla_j Basis_k - \nabla_j \nabla_i Basis_k - \nabla_{[Basis_i,\, Basis_j]} Basis_k = {Curv_k}^l{}_{ij} Basis_l.$$

The second structure equation relates the exterior derivatives of the connection forms to curvature forms. Like the first structure equation, it is applied only when you specifically request it by applying `SecondStructureRule`:

```
                  j           k          j
Out[15]= d[Conn     ] - Conn     ^ Conn
                i           i          k

In[16]:= % /. SecondStructureRule //TensorSimplify

         1      j
Out[16]= - Curv
         2     i
```

As described above for `Tor`, the factor 1/2 appears when `$WedgeConvention = Alt`, but not when `$WedgeConvention = Det`.

You can cause components of the curvature tensor `Curv` to be expanded in terms of components of `Conn` and their directional derivatives by applying the Ricci rule `CurvToConnRule`.

You can apply all three structure equations (`CompatibilityRule`, `FirstStructureRule`, and `SecondStructureRule`) at once by using the rule `StructureRules`.

If an expression involving derivatives of connection forms is converted to a component expression, the derivatives of the connection coefficients are maintained in a form such as

```
               j
Del   [Conn     ]
   k       i
```

since "covariant derivatives" of connection coefficients make no invariant sense. This is the only instance in which Ricci does not by default express components of derivatives in terms of covariant derivatives.

7.6 Non-default connections and metrics

You may want to compute covariant derivatives with respect to a connection or a metric other than the default one on a given bundle. Many of Ricci's differentiation commands accept the `Connection` option to indicate that covariant derivatives are to be taken with respect to a non-default connection. Ordinarily, the value of the `Connection` option should be an expression of the form `Conn + diff`, where `diff` is a 3-tensor expression that represents the "difference tensor" between the given connection and the default connection. For example, suppose `diff` is a 3-tensor:

```
In[23]:= Del[a, Connection -> Conn + diff]

            (Conn + diff)
Out[23]= Del             [a]

In[24]:= %[L[i],L[j],L[k]]

                                     k36                     k37
Out[24]= a         - a        diff          - a        diff
          i j ;k      k36 j       i      k     i k37       j     k
```

Notice that the components are expanded in terms of covariant derivatives of `a` (with respect to the default connection) and components of `diff`.

You can also call `CovD` with the `Connection` option. For example, the following input expression results in the same output as `Out[24]` above:

```
In[25]:= CovD[ a[L[i],L[j]], {L[k]}, Connection -> Conn + diff]
```

Other operators that accept the `Connection` option are `Div`, `Grad`, and `Laplacian`.

On a Riemannian tangent bundle, you can also specify that operations are to be computed with respect to a metric other than the default one. This is done by adding the `Metric` option to the appropriate Ricci function. For example, suppose `h` is a symmetric 2-tensor. To compute the covariant derivative of the tensor `a` with respect to the metric `h`, you could use

```
In[60]:= Del[ a , Metric -> h]

               (h)
Out[60]= Del      [a]
```

When you insert indices, the components of this expression are computed in terms of covariant derivatives of `a` and `h` (with respect to the default metric). Other differentiation operators that accept the `Metric` option are `Grad`, `Div`, `ExtdStar`, `Laplacian`, `LaplaceBeltrami`, and `CovD`.

Ricci can explicitly compute the Riemannian, Ricci, and scalar curvatures of a non-default metric `h`. These are indicated by the expressions `RiemannTensor[h]`, `RicciTensor[h]`, and `ScalarCurv[h]`, respectively. When you insert indices into these, you get expressions for the curvatures in terms of the curvatures of the default metric and covariant derivatives of `h`. You can also refer to the Levi-Civita connection of a non-default metric by `LeviCivitaConnection[h]`; this produces a 3-tensor that is suitable as an input to the `Connection` option of the differentiation functions as described above.

8 Reference List

8.1 Ricci commands and functions

The list below describes all Ricci functions that are accessible to users. All the essential information about each function is collected here in brief form, including a list of all available options. The numbers at the right-hand sides of the function descriptions refer to section numbers in the main text where these functions are described. (Some less often used functions are documented only here in the Reference List.)

For more information and examples on how to use the most common commands, see the main text. For descriptions of Ricci's global variables (symbols starting with `$`), see the end of this chapter.

`AbsorbMetrics` 5.4

`AbsorbMetrics[x]` simplifies x by eliminating any metrics that are contracted with other tensors, and using them instead to raise or lower indices. `AbsorbMetrics[x,n]` or `AbsorbMetrics[x,{n1,n2,...}]` applies `AbsorbMetrics` only to term `n` or to terms `n1,n2,...` of `x`.

Option:

- `Mode -> All` or `NoOneDims`. `NoOneDims` indicates that metrics associated with one-dimensional bundles should not be absorbed, unless they are paired with other metrics. This option exists mainly for internal use by `TensorSimplify`, so that it can control the order in which different metrics are absorbed. Default is `All`.

See also `TensorSimplify`, `LowerAllIndices`, `CorrectAllVariances`.

`Alt` 3.8

`Alt[x]` is the alternating part of the tensor expression `x`. If `x` is alternating, then `Alt[x] = x`.

`Alt` is also a value for the global variable `$WedgeConvention`.

See also `Sym`, `Wedge`, `$WedgeConvention`.

`Alternating` 2.4

`Alternating` is a value for the `Symmetries` option of `DefineTensor`.

`Any` 2.1

`Any` can be used as a value for the `Bundle` option of the `DefineTensor` command. `Bundle -> Any` means that indices from any bundle can be inserted.

`Bar` 2.2, 2.4

`Bar` is an internal symbol used by Ricci for representing conjugates of tensors and indices. The internal form for a barred index is `L[i[Bar]]` or `U[i[Bar]]`. In input form, these can be abbreviated `LB[i]` and `UB[i]`. The internal form for the conjugate of a tensor is `Tensor[name[Bar],{...},{...}]`. In input form, this is typed `Conjugate[name] [...] [...]`. See also `Tensor`, `Conjugate`, `L`, `U`, `LB`, `UB`.

`Basis` 2.4, 5.10

`Basis` is the name used for generic basis vectors and covectors for any bundle. Basis vectors are generated automatically by `BasisExpand`. For example, if index name `i` is associated with bundle `b`, then `Basis[L[i]]` and `Basis[U[i]]` represent contravariant and covariant basis elements for `b` (i.e., basis elements for `b` and its dual), respectively; `Basis[LB[i]]` and `Basis[UB[i]]` represent basis elements for the conjugate bundle `Conjugate[b]` and its dual.

`Basis` is defined by Ricci as a product tensor of rank {1,1}, with the special option `Bundle -> Same`: this means that the first index can be associated with any bundle, but when a second index is inserted, it must be associated with the same bundle as the first, or else the result is 0. If you insert an index into `Basis[L[i]]` or `Basis[U[i]]`, you get Kronecker delta coefficients, metric coefficients (which result from lowering or raising an index on the Kronecker delta), or 0 as appropriate. See also `BasisExpand`, `BasisName`, `CoBasisName`, `Kronecker`.

`BasisExpand` 2.4, 5.10

`BasisExpand[x]` expands all tensors appearing in `x` into component expressions multiplied by basis vectors and covectors, `Basis[L[i]]` and `Basis[U[j]]`. `BasisExpand[x,n]` or `BasisExpand[x,{n1,n2,...}]` applies `BasisExpand` only to term `n` or to terms `n1,n2,...` of `x`. `BasisExpand` also inserts indices into all unfilled index slots, so it can be used to turn a scalar tensor expression into a component expression. See also `Basis`, `BasisGather`, `CovDExpand`, `ProductExpand`.

`BasisGather` 5.10

`BasisGather[x,y]` attempts to recognize the basis expression for `y` in `x`, and replaces it by `y`. `BasisGather[ x, {y1,y2,...} ]` does the same thing for several expressions at once. In effect, `BasisGather` is a partial inverse for `BasisExpand`. See also `BasisExpand`.

BasisName 2.4

`BasisName -> name` is an option for `DefineBundle`. It specifies the name to be used in input and output form for the basis vectors for the bundle being defined.

BianchiRules 7.2

`BianchiRules[L[i],L[j],L[k]]` is a rule that converts Riemannian curvature tensors containing the indices `L[i]`, `L[j]`, `L[k]` in order to two-term sums, using the first and second Bianchi identities. The three indices may be upper indices, lower indices, or a combination. The first Bianchi rule is applied to any tensor with with `Symmetries -> RiemannSymmetries`; the second is applied only to tensors that have been defined as Riemannian curvature tensors by `DefineBundle`.

EXAMPLES:

```
            i
Out[16]= Rm
              j k l

In[17]:= % /. BianchiRules[L[j],L[k],L[l]]

            i             i
Out[17]= Rm          - Rm
              k j l         l j k


                     m
Out[20]= Rm
            i j k l ;

In[21]:= %/.BianchiRules[L[k],L[l],U[m]]

                  m                m
Out[21]= Rm               - Rm
            i j k   ;l        i j l   ;k
```

See also `DefineBundle`, `FirstBianchiRule`, `SecondBianchiRule`, `ContractedBianchiRules`.

Bundle 2.4

`Bundle -> name` is a `DefineTensor` option, specifying the name of the bundle or bundles the tensor is to be associated with.

The function `Bundle[i]` returns the name of the bundle associated with the index `i`.

`BundleDummy`

`BundleDummy[bundle]` returns the symbol that is used for computer-generated dummy indices associated with the bundle. This is the last index name in the list `BundleIndices[bundle]`. For most bundles, new dummy indices are of the form `k`n, where `k=BundleDummy[bundle]` and n is an integer. For one-dimensional bundles, n is omitted. See also `BundleIndices`, `DefineBundle`.

`BundleIndices`

`BundleIndices[bundle]` returns a list of the index names currently associated with bundle. See also `BundleDummy`, `DefineBundle`.

`BundleQ`

`BundleQ[x]` returns `True` if `x` is the name of a bundle, and `False` otherwise. See also `DefineBundle`.

`Bundles`

The function `Bundles[x]` returns a list of the bundles associated with each index position in the tensor expression `x`; each entry in the list is a sublist giving the allowed bundles for that index position. If you insert an index that is not associated with one of the bundles corresponding to that index position, the expression gives 0. See also `DefineTensor`, `Variance`.

`Co` 2.4

`Co` is an abbreviation for `Covariant`.

`CoBasisName` 2.4

`CoBasisName -> name` is an option for `DefineBundle`. It specifies the name to be used in input and output form for the basis covectors for the bundle being defined.

`CollectConstants` 5.7

`CollectConstants[x]` groups together terms in the tensor expression `x` having the same tensor factors but different constant factors. `CollectConstants[x,n]` or `CollectConstants[x,{n1,n2,...}]` applies `CollectConstants` only to term `n` or to terms `n1,n2,...` of `x`. The constant factors are simplified somewhat: if a constant factor is not too complicated (`LeafCount[c]` $\leq$ 50), Ricci factors the constant. For larger constant expressions, it applies the Mathematica function `Together`. See also `TensorSimplify`, `FactorConstants`, `SimplifyConstants`, `ConstantFactor`, `TensorFactor`.

`CommuteCovD` 5.17

`CommuteCovD[ x, L[i], L[j] ]` changes all adjacent occurrences of indices `L[i],L[j]` after the ";" to `L[j],L[i]` by adding appropriate curvature and torsion terms. The second and third arguments may be upper or lower indices. See also `OrderCovD`, `CovDSimplify`.

`CommutingFrame` 2.1

`CommutingFrame` is a `DefineBundle` option. Default is `True`.

`CompatibilityRule` 7.5

`CompatibilityRule` is a rule that transforms derivatives of metric components into connection coefficients, using the fact that the default connection is compatible with the metric, which is expressed by

$$d(g_{ij}) = Conn_{ij} + Conn_{ji}.$$

See also `FirstStructureRule`, `SecondStructureRule`, `StructureRules`.

`Complex` 2.1, 2.3, 2.4, 2.6

Ricci recognizes `Complex` as a value for the `Type` option of `DefineConstant`, `DefineTensor`, `DefineBundle`, and `DefineMathFunction`.

`Con` 2.4

`Con` is an abbreviation for `Contravariant`.

`Conjugate` 2.1, 2.2, 2.4, 2.6

The Ricci package modifies the Mathematica function `Conjugate` to handle tensor expressions and indices. The behavior of a tensor, constant, index, bundle, or mathematical function under conjugation is determined by the `Type` option specified when the object is defined.

In input form, you indicate the conjugate of tensor `t` by typing `Conjugate[t]`; you may follow this with one or more sets of indices in brackets, as in `Conjugate[t] [L[i],L[j]]`. You indicate the conjugate of an index `L[i]` or `U[i]` by typing `LB[i]` or `UB[i]`. To indicate the conjugate of bundle `b`, type `Conjugate[b]`. You can compute the conjugate of any tensor expression `x` by typing `Conjugate[x]`. In output form, conjugates of tensors, bundles, and indices appear as barred symbols.

The behavior of tensors and indices under conjugation is determined by the `Type` options specified when the tensor and its associated bundles are defined. For example, assume `a`, `b`, and `c` are 2-tensors of types `Real`, `Complex`, and `Imaginary`, respectively. Assume `i` is an index associated with a real bundle, and `p` is an index associated with a complex bundle:

```
Conjugate[a    ] =  a  _
           i p       i p
                    _
Conjugate[b    ] =  b  _
           i p       i p

Conjugate[c    ] = -c  _
           i p       i p
```

See also `Re`, `Im`, `DefineTensor`, `DefineConstant`, `DefineBundle`, `Declare`.

`Conn` 7.5, 7.6

`Conn` is the name used for the generic connection forms for the default connection on any bundle. It is defined as a product tensor of rank {2,1} and variance {`Covariant`, `Contravariant`, `Covariant`}. When the first two indices are inserted, it represents an entry of the matrix of connection 1-forms associated with the default basis. When all three indices are inserted, it represents the Christoffel symbols of the default connection relative to the default basis. The upper index is the second one, and the index representing the direction in which the derivative is taken is the third one. Thus the Christoffel symbol Γ^k_{ij} is represented by

```
                                   k
Conn [L[j],U[k],L[i]]  -->  Conn
                                 j   i
```

See also `Curv`, `Tor`, `Curvature`, `Connection`, `LeviCivitaConnection`, `CovDExpand`, `ConnToMetricRule`, `CurvToConnRule`.

`Connection` 7.6

`Connection -> cn` is an option for `Del`, `CovD`, `Div`, `Grad`, and `Laplacian`. It specifies that covariant derivatives are to be taken with respect to the connection `cn` instead of the default connection.

`ConnToMetricRule` 7.2

`ConnToMetricRule` is a rule that causes indexed components of the default connection `Conn` on a Riemannian tangent bundle to be converted to expressions involving directional derivatives of the metric coefficients. This rule applies only to fully-indexed connection components whose indices all refer to a single Riemannian tangent bundle with the option `CommutingFrame -> True`. See also `CurvToConnRule`, `Conn`, `DefineBundle`.

`ConstantFactor`

`ConstantFactor[x]` returns the product of all the constant factors in `x`, which should be a monomial. See also `TensorFactor`, `CollectConstants`.

`ConstantQ` 2.3

`ConstantQ[x]` returns `True` if there are no explicit tensors in `x`, and `False` otherwise. See also `DefineConstant`.

`ContractedBianchiRules` 7.2

`ContractedBianchiRules` is a rule that simplifies contracted covariant derivatives of the Riemannian and Ricci curvature tensors using contracted versions of the second Bianchi identity. For example (assuming `$RiemannConvention = SecondUp`, which is the default),

```
                     l
Out[3]= Rm
           i j k l ;

In[4]:= % /. ContractedBianchiRules

Out[4]= -Rc         + Rc
            i k ;j       j k ;i
```

Similarly,

```
               j
Out[5]= Rc
           i j ;

In[6]:= % /. ContractedBianchiRules

        1
Out[6]= - Sc
        2   ;i
```

See also `BianchiRules`, `FirstBianchiRule`, `SecondBianchiRule`.

`Contravariant` 2.4

`Contravariant` is a value for the `Variance` option of `DefineTensor`, and may be abbreviated `Con`. If an index slot is `Contravariant`, indices in that slot are upper by default. See also `DefineTensor`.

`CorrectAllVariances` 5.15

`CorrectAllVariances[x]` changes the variances (upper to lower or lower to upper) of indices in `x` whose variances are not correct for their positions, by inserting appropriate metric coefficients. `CorrectAllVariances[x,n]` or `CorrectAllVariances[x,{n1,n2,...}]` applies `CorrectAllVariances` only to term `n` or to terms `n1,n2,...` of `x`.

Option:

- `Mode -> All` or `OneDims`. `OneDims` indicates that only one-dimensional indices and indices that appear inside of differential operators such as `Del` or `Extd` should be corrected. This option exists mainly for internal use by `TensorSimplify`. Default is `All`.

See also `TensorSimplify`, `LowerAllIndices`, `AbsorbMetrics`.

`Covariant` 2.4

`Covariant` is a value for the `Variance` option of `DefineTensor`, and may be abbreviated `Co`. If an index slot is `Covariant`, indices in that slot are lower by default. See also `DefineTensor`.

`CovD` 4.2

If `x` is a rank-0 tensor expression, then `CovD[ x, {L[i],L[j]} ]` is the component of the covariant derivative of `x` in the `L[i],L[j]` directions. If `x` has any unfilled index slots, then it is first converted to a component expression by generating dummy indices if necessary before the covariant derivative is computed. The shorthand input form `x[ L[i], L[j] ]` is automatically converted to `CovD[ x, {L[i],L[j]} ]` if `x` has rank 0. In output form, covariant derivatives of a tensor are represented by subscripts following a semicolon.

Options (for the explicit form `CovD[x,{L[i],L[j]}]` only):

- `Connection -> cn` specifies that covariant derivatives are to be taken with respect to the connection `cn` instead of the default connection. Ordinarily, `cn` should be an expression of the form `Conn + diff`, where `diff` is a 3-tensor expression representing the "difference tensor" between the default connection and `cn`.

- `Metric -> g`: a symmetric 2-tensor expression, indicating that the covariant derivatives are to be taken with respect to the Levi-Civita connection of `g` instead of the default metric for `x`'s bundle (which is assumed to be Riemannian).

See also `Del`, `InsertIndices`, `CovDExpand`.

`CovDExpand` 5.12

`CovDExpand[x]` converts all components of covariant derivatives in `x` to ordinary directional derivatives and connection coefficients. `CovDExpand[x,n]` or `CovDExpand[x,{n1,n2,...}]` applies `CovDExpand` only to term `n` or to terms `n1,n2,...` of `x`. Directional derivatives with respect to basis elements are represented by `Del[ Basis[L[i]],...]`. For example:

```
                                                k1
 v      // CovDExpand --> Del  [v  ] - Conn       v
  i ;j                       j   i          j    i  k1
```

Ordinary directional derivatives can be converted back to covariant derivatives by calling `BasisExpand`. See also `BasisExpand`, `Del`, `Conn`.

`CovDSimplify` 5.19

`CovDSimplify[x]` attempts to simplify `x` as much as possible by ordering all dummy indices, including those that occur after the ";" in component expressions. `CovDSimplify[x,n]` or `CovDSimplify[x,{n1,n2,...}]` applies `CovDSimplify` only to term `n` or to terms `n1,n2,...` of `x`. `CovDSimplify` first calls `TensorSimplify`, then `OrderCovD`, then `TensorSimplify` again. See also `OrderCovD`, `CommuteCovD`, `TensorSimplify`, `SuperSimplify`.

`Curv` 7.5, 7.6

`Curv` is the generic curvature tensor associated with the default connection on any bundle. It is generated when covariant derivatives are commuted, and when `SecondStructureRule` is applied to derivatives of connection forms. `Curv` is a product tensor with rank {2,2}, which is `Alternating` in its first two and last two indices. Inserting the first two indices yields the matrix of curvature 2-forms associated with the default basis. Inserting all four indices yields the coefficients of the curvature tensor. See also `Conn`, `Curvature`, `CurvToConnRule`.

`Curvature` 7.5

`Curvature[cn]` is the curvature tensor associated to the connection `cn`. Ordinarily, `cn` should be an expression of the form `Conn + diff`, where `diff` is a 3-tensor expression representing the "difference tensor" between the default connection and `cn`. See also `Conn`, `Curv`, `CurvToConnRule`.

`CurvToConnRule` 7.2, 7.5

`CurvToConnRule` is a rule that converts components of curvature tensors (including Riemann, Ricci, and scalar curvature tensors for Riemannian tangent bundles) to expressions involving connection coefficients and their covariant derivatives. It is a partial inverse for `SecondStructureRule`. See also `Curv`, `Conn`, `ConnToMetricRule`, `SecondStructureRule`.

`Declare` 2.1, 2.3, 2.4

`Declare[ name, options ]` can be used to change certain options for previously-defined Ricci objects. `Declare[ {name1,name2,...}, options ]` changes options for several names at once; all names in the list are given the same options. The first argument, `name`, must be the name of a bundle, constant, index, tensor, or mathematical function that was previously defined by a call to `DefineBundle`, `DefineConstant`, `DefineIndex`, `DefineTensor`, or `DefineMathFunction`. The allowable options depend on what type of object is given as the first argument; the meanings of the options are the same as in the corresponding `Define` command. Any options not listed explicitly in the call to `Declare` are left unchanged.

- *Constant:* If `name` is a symbolic constant, only the `Type` option is allowed. It overrides any `Type` specification given when `DefineConstant` was called.
- *Index:* If `name` is an index name, the only allowable option is `TeXFormat`.
- *Bundle:* If `name` is a bundle name, the allowable options are `FlatConnection`, `ParallelFrame`, `OrthonormalFrame`, `PositiveDefinite`, `CommutingFrame`, and `TorsionFree`.
- *Tensor:* If `name` is a tensor name, the allowable options are `Type`, `TeXFormat`, and `Bundle`.
- *Math function:* If `name` is the name of a mathematical function, the only allowable option is `Type`.

See also `DefineBundle`, `DefineConstant`, `DefineIndex`, `DefineTensor`, `DefineMathFunction`.

`DefineBundle` 2.1, 7.1

`DefineBundle[ name, dim, metric, {indices} ]` defines a bundle.

- `name`: A symbol that uniquely identifies the bundle.

- `dim`: The dimension of the bundle. Can be a positive integer, a symbolic constant, or any constant expression.
- `metric`: A name for the bundle's metric. The metric is automatically defined by `DefineBundle` as a `Symmetric` or `Hermitian` 2-tensor. It is assumed to be compatible with the bundle's default connection.
- {`indices`}: A list of index names to be associated with the bundle. If only one index name is given, the braces may be omitted. For one-dimensional bundles, only one index name is allowed. For higher-dimensional bundles, additional indices can be added to the list at any time by calling `DefineIndex`.

Options:

- `Type -> Complex` or `Real`. Default is `Real`. If `Complex`, the conjugate bundle is referred to as `Conjugate[name]`, and is associated with barred indices.
- `TangentBundle -> bundle`. Specifies the name of the underlying tangent bundle for the bundle being defined. This can be a list of bundles, representing the direct sum of the bundles in the list. The default is the value of `$DefaultTangentBundle` if defined, otherwise this bundle itself (or the direct sum of this bundle and its conjugate if the bundle is complex). If `$DefaultTangentBundle` has not been given a value the first time `DefineBundle` is called, it is set to the first bundle defined (and its conjugate if it is a complex bundle).
- `FlatConnection -> True` or `False`. `True` means the default connection on this bundle has curvature equal to 0. The default is `False`.
- `ParallelFrame -> True` or `False`. `True` means the default basis elements have vanishing covariant derivatives, so the connection and curvature forms are set equal to 0. This option forces `FlatConnection -> True`. The default is `False`.
- `OrthonormalFrame -> True` or `False`. `True` means the metric coefficients with respect to the default basis are always assumed to be constants. For one-dimensional bundles, the metric coefficient is explicitly set to 1. The default is `False`.
- `CommutingFrame -> True` or `False`. This option is meaningful only for tangent bundles or their subbundles. `True` means the default basis for this bundle consists of commuting vector fields (as is the case, for example, when coordinate vector fields are used). This option causes the connection coefficients for this bundle to be symmetric in their two lower indices, and Lie brackets of basis vectors to be replaced by zero.

- `PositiveDefinite -> True` or `False`. `True` means the bundle's metric is positive definite. `False` means no assumption is made about the sign of the metric. This affects the simplification of complex powers and logarithms of inner products with respect to the metric. The default is `True`.
- `TorsionFree -> True` or `False`. `False` means that the default connection on this bundle may have nonvanishing torsion; this makes sense only for tangent bundles or their subbundles. The default is `True`.
- `BasisName -> name`. Specifies a name to be used in input, output, and TEX forms for the default basis vectors for this bundle. The default name is `Basis`. Basis vectors are always represented internally by the name `Basis`. Covariant basis vectors can be named independently by the `CoBasisName` option.
- `CoBasisName -> name`. Specifies a name to be used in input, output, and TEX forms for the default covariant basis vectors for this bundle. The default is the name given in the `BasisName` option, or `Basis` if `BasisName` is also omitted. Basis covectors are always represented internally by the name `Basis`.
- `MetricType -> Riemannian` or `Normal`. The default is `Normal`, which means that the bundle's metric has no special properties. If `Riemannian` is specified, this bundle is assumed to be a Riemannian tangent bundle, and `DefineBundle` automatically defines the Riemannian, Ricci, and scalar curvature tensors. Contractions of the Riemannian and Ricci curvatures are automatically converted to Ricci and scalar curvatures, respectively. To apply other identities, use `FirstBianchiRule`, `SecondBianchiRule`, `ContractedBianchiRules`, or `BianchiRules`.

  When you specify `MetricType -> Riemannian`, you may also give the `RiemannTensor`, `RicciTensor`, `ScalarCurv`, and `RiemannConvention` options below.
- `RiemannTensor -> name`: a name to be given to the Riemannian curvature tensor for this bundle. This tensor is automatically defined as a real, covariant 4-tensor with `Symmetries -> RiemannSymmetries`. This option takes effect only if `MetricType -> Riemannian` is specified. The default is `Rm`.
- `RicciTensor -> name`: a name to be given to the Ricci tensor for this bundle. This tensor is automatically defined as a real, covariant, symmetric 2-tensor. This option takes effect only if `MetricType -> Riemannian` is specified. The default is `Rc`.
- `ScalarCurv -> name`: a name to be given to the scalar curvature function for this bundle. This tensor is automatically defined as

a real 0-tensor. This option takes effect only if `MetricType -> Riemannian` is specified. The default is `Sc`.

- `RiemannConvention -> SecondUp` or `FirstUp`: determines the sign convention used for the Riemannian curvature tensor for this bundle. This option takes effect only if `MetricType -> Riemannian` is specified. The default is `SecondUp`, or the value of the global variable `$RiemannConvention` if it has been set. The meaning of this option is indicated by the relations below. (Note that the Riemann curvature tensor is always defined as a *covariant* 4-tensor, so all four of its indices are normally down. It is done this way so that all index positions are equivalent for the purpose of applying symmetries of the curvature tensor. The `RiemannConvention` option controls only the sign of the curvature components.)

  `RiemannConvention -> FirstUp`:

$$(\nabla_X \nabla_Y - \nabla_Y \nabla_X - \nabla_{[X,Y]})Z = X^i Y^j Z^k g^{\ell m} Rm_{mkij} Basis_\ell$$

$$Rm^k{}_{ikj} = Rc_{ij}$$

  `RiemannConvention -> SecondUp`:

$$(\nabla_X \nabla_Y - \nabla_Y \nabla_X - \nabla_{[X,Y]})Z = X^i Y^j Z^k g^{\ell m} Rm_{kmij} Basis_\ell$$

$$Rm_i{}^k{}_{kj} = Rc_{ij}$$

- `Quiet -> True` or `False`. Default is `False`, or the value of `$Quiet` if set.

See also `DefineTensor`, `DefineIndex`, `UndefineBundle`, `Declare`.

`DefineConstant` 2.3

`DefineConstant[c]` defines the symbol `c` to be a constant with respect to differentiation in all space variables. The notation `c[ L[i] ]` is always interpreted as a covariant derivative of `c`, and thus 0; and `Conjugate[c]` is replaced by `c` if the constant is `Real`, and by `-c` if it is `Imaginary`.

Options:

- `Type -> type`. This can be a single keyword or a list of keywords, chosen from among `Complex`, `Real`, `Imaginary`, `Positive`, `Negative`, `NonPositive`, `NonNegative`, `Integer`, `Odd`, or `Even`. Determines how the constant behaves under conjugation, exponentiation, and logarithms. Default is `Real`.
- `Quiet -> True` or `False`. Default is `False`, or the value of `$Quiet` if set.

See also `Conjugate`, `UndefineConstant`, `Declare`.

`DefineIndex` 2.1, 2.2

`DefineIndex[ {i,j,k}, bundle ]` causes the index names `i`, `j`, `k` to be associated with bundle.

- The first argument must be a symbol or list of symbols. One or more of these may have already been defined as indices, as long as they are associated with the same bundle as specified in this command.
- The second argument must be the name of a bundle.

Options:

- `TeXFormat -> "texformat"`. Determines how the index appears in `TeXForm` output. Default is the index name itself. This should be specified only when one single index is being defined. Note that, as in all Mathematica strings, backslashes must be typed as double backslashes; thus to get an index name to appear as β in TEX output, you would type `TeXFormat -> "\\beta"`.
- `Quiet -> True` or `False`. Default is `False`, or the value of `$Quiet` if set.

See also `DefineBundle`, `UndefineIndex`, `UndefineBundle`, `Declare`.

`DefineMathFunction` 2.5, 2.6

`DefineMathFunction[f]` declares `f` to be a scalar-valued function of one real or complex variable that can be used in tensor expressions. If `x` is any rank-0 tensor expression, then `f[x]` is also interpreted as a rank-0 tensor expression. Ricci has built-in definitions for the Mathematica functions `Sin`, `Cos`, `Tan`, `Sec`, `Csc`, `Cot`, and `Log`. If you intend to use any other functions in tensor expressions, you must call `DefineMathFunction`.

LIMITATION: Currently, Ricci has no provision for using mathematical functions depending on more than one variable in tensor expressions.

Options:

- `Type -> type`. This can be a single keyword or a list of keywords, chosen from among `Real`, `Complex`, `Imaginary`, `Automatic`, `Positive`, `Negative`, `NonPositive`, or `NonNegative`. `Real` means `f[x]` is always real; `Imaginary` means `f[x]` is always imaginary; `Complex` means `f[x]` can assume arbitrary complex values; and `Automatic` means `Conjugate[f[x]] = f[Conjugate[x]]`. `Positive`, `Negative`, `NonPositive`, and `NonNegative` all imply `Real`, and mean that the function values always have the corresponding attribute. Default is `Real`.

- `Quiet -> True` or `False`. Default is `False`, or the value of `$Quiet` if set.

See also `Declare`.

`DefineRelation` 6.1

`DefineRelation[ tensor, x, condition ]` defines a relation of the form

```
tensor := x /; condition
```

associated with the tensor's name. The first argument `tensor` must be a single tensor name with or without indices, and can be written in Ricci's input form. If index names used in `tensor` are associated with bundles, then the relation is applied only when indices from those bundles appear in those positions. The `condition` argument is optional; if present, it should be a true/false condition that must be satisfied before the relation will be applied. It may involve the tensor name or indices that appear in `tensor`.

For example, if `i` is associated with bundle `a` and `p` with bundle `b`,

```
DefineRelation[ t[ L[i],L[p] ], 0 ]
```

specifies that `t[ _, _ ]` is to be replaced by `0` whenever its first index is associated with `a` and its second with `b`. In general, the relation substitutes `x` (suitably modified) in place of any expression that matches `tensor` after insertion of indices, renaming of indices, covariant differentiation, conjugation, or raising or lowering of indices.

`DefineRelation` has the `HoldAll` attribute, so its arguments will not be evaluated when the command is executed. This means you must write out `tensor`, `x`, and `condition` in full, rather than referring to a variable name or a preceding output line such as `Out[10]`, so that `DefineRelation` can properly process the indices that appear in the arguments.

Option:

- `Quiet -> True` or `False`. Default is `False`, or the value of `$Quiet` if set.

See also `DefineRule`, `UndefineRelation`.

`DefineRule` 6.2

`DefineRule[ rulename, lhs, rhs, condition ]` defines a rule named `rulename` of the form

```
lhs :> rhs /; condition
```

The first two arguments, `lhs` and `rhs`, are arbitrary tensor expressions with or without indices; tensors in them can be written in Ricci's input form (tensor name, followed optionally by one or more sets of indices in brackets). Other expressions used in `lhs` must match Mathematica's internal form for such expressions. The `condition` argument is optional; if present, it should be a true/false condition that must be satisfied before the relation will be applied. It may involve the tensor name or indices that appear in `lhs`.

The rule is appended to a list containing previous rules defined with the same name, unless the option `NewRule -> True` is specified. To delete a rule entirely, make the assignment

```
rulename = .
```

If index names used in `lhs` are associated with bundles, then the rule is applied only when indices from those bundles appear in those positions. For example, if `i` is associated with bundle `a` and `p` with bundle `b`,

```
DefineRule[ zerorule, t[ L[i], L[p] ], 0 ]
```

causes rule `zerorule` to replace `t[ _, _ ]` by `0` whenever its first index is associated with `a` and its second with `b`. In general, the rule substitutes `rhs` (suitably modified) in place of any expression that matches `lhs` after renaming of indices or raising or lowering of indices. If `lhs` is a single tensor with or without indices, then the rule is also applied to expressions that match `lhs` after insertion of indices, covariant differentiation, or conjugation.

`DefineRule` has the `HoldAll` attribute, so its arguments will not be evaluated when the command is executed. This means you must write out `lhs`, `rhs`, and `condition` in full, rather than referring to a variable name or a preceding output line such as `Out[10]`, so that `DefineRule` can properly process the indices that appear in the arguments.

Options:

- `Quiet -> True` or `False`. Default is `False`, or the value of `$Quiet` if set.
- `NewRule -> True` or `False`. Default is `False`.

See also `DefineRelation`.

`DefineTensor` 2.4, 7.4

`DefineTensor[ name, rank ]` defines a tensor.

- `name`: A symbol that uniquely identifies the tensor.
- `rank`: Rank of the tensor. For ordinary tensors, this must be a non-negative integer. For product tensors, this is a list of positive integers.

Options:

- `Symmetries -> sym`: Symmetries of the tensor. For ordinary tensors, the allowed values are `Symmetric`, `Alternating` or `Skew`, `Hermitian`, `SkewHermitian`, `RiemannSymmetries`, `NoSymmetries`, or the name of an arbitrary symmetry defined by calling `DefineTensorSymmetries`. For a product tensor, the value of the `Symmetries` option must be either `NoSymmetries` or a list of symmetries whose length is the same as that of the list of ranks; each symmetry on the list can be `Symmetric`, `Alternating` or `Skew`, `Hermitian`, `SkewHermitian`, or `NoSymmetries`, and applies to the corresponding group of indices in the list of ranks. Default is `NoSymmetries`.

  The values of the `Symmetries` option have the following meanings:

  `Symmetric`: The tensor is fully symmetric in all indices.

  `Alternating`: The tensor changes sign if any two indices are interchanged. `Skew` is a synonym for `Alternating`.

  `Hermitian`: The tensor must be a real 2-tensor associated with a bundle and its conjugate. It is symmetric in its two indices, with the additional property that `t[L[i],L[j]]` and `t[LB[i],LB[j]]` are both zero whenever `i,j` are associated with the same bundle. For product tensors, this can be used only for a group of two indices.

  `SkewHermitian`: The tensor must be a real 2-tensor associated with a bundle and its conjugate. It is skew-symmetric in its two indices, with the additional property that `t[L[i],L[j]]` and `t[LB[i],LB[j]]` are both zero whenever `i,j` are associated with the same bundle. For product tensors, this can be used only for a group of two indices.

  `RiemannSymmetries`: The tensor must have rank 4. The symmetries are those of the Riemann curvature tensor: skew-symmetric in the first two indices, skew-symmetric in the last two indices, symmetric when the first two and last two are interchanged. The (first) Bianchi identity is not automatically applied, but can be explicitly applied to any tensor with these symmetries by means of `FirstBianchiRule` or `BianchiRules`.
- `Type -> type`. For rank-0 tensors, this can be a single keyword or a list of keywords, chosen from among `Complex`, `Real`, `Imaginary`,

`Positive`, `Negative`, `NonPositive`, or `NonNegative`. For higher-rank tensors, only `Real`, `Complex`, or `Imaginary` is allowed. Determines how the tensor behaves under conjugation, exponentiation, and logarithms. The default is always `Real`.

- `TeXFormat -> "texformat"`: Determines how the tensor appears in `TeXForm` output. Default is the tensor's name. Note that, as in all Mathematica strings, backslashes must be typed as double backslashes; thus to get a tensor's name to appear as β in TEX output, you would type `TeXFormat -> "\\beta"`.

- `Bundle -> bundle`: The bundle with which the tensor is associated. This can be a list, meaning the tensor is associated with the direct sum of all the bundles in the list. If a complex bundle is named and the `Type` option is not `Complex`, the tensor is automatically associated with the direct sum of the bundle and its conjugate.

  `Bundle -> Any` indicates that the tensor can accept indices from any bundle.

  The default value for the `Bundle` option is `$DefaultTangentBundle`.

  If the tensor is given an index associated with a bundle not in this list, the result is 0.

  For a product tensor, the `Bundle` option can be either a single bundle, meaning that all index positions are associated with the same bundle, or a list whose length is the same length as the list of ranks. Each entry in the list specifies the bundle for the corresponding index slots. Each entry can be either a single bundle name or a list of bundle names indicating a direct sum of bundles for that group of slots.

  `Bundle -> Same` is a special option for product tensors. It means that indices from any bundle can be inserted in the first set of index slots, but indices in the remaining slots must be from the same bundle or its conjugate. This option is used internally by Ricci in defining the `Basis` tensor.

  `Bundle -> Automatic` is another special form that can be used for product tensors. It means that indices from any bundle can be inserted in the first set of index slots, but indices in the remaining slots must come from the underlying tangent bundle of the first indices. This option is used internally by Ricci in defining the `Conn`, `Tor`, and `Curv` tensors.

- `Variance -> Covariant` or `Contravariant`: Specifies whether the tensor is covariant (so that components have lower indices) or contravariant (upper indices). This can be a list whose length is equal to the total rank of the tensor, in which case each entry in the list specifies the variance of the corresponding index slot. The default is `Covariant`. Values may be abbreviated `Co` and `Con`.

- `Quiet -> True` or `False`: Default is `False`, or the value of `$Quiet` if set.

See also `UndefineTensor`, `Declare`, `Conjugate`, `DefineBundle`, `FirstBianchiRule`, `BianchiRules`, `Conn`, `Tor`, `Curv`, `Basis`, `DefineTensorSymmetries`.

`DefineTensorSymmetries`

`DefineTensorSymmetries[ name, {perm1,sgn1,...,permN,sgnN} ]` defines a `TensorSymmetry` object that can be used in `DefineTensor`. Here `perm1,...,permN` must be lists that are nontrivial permutations of `{1,2,...,k}`, and `sgn1,...,sgnN` must be constants (ordinarily ± 1). A tensor with this symmetry has the property that, if any of the permutations `perm1,...,permN` is applied to its indices, the value of the tensor is multiplied by the corresponding constant `sgn1,...,sgnN`. Ricci automatically tries all non-trivial iterates of each specified permutation, until the indices are as close as possible to lexical order. (However, Ricci does not try composing different permutations in the list.) See also `TensorSymmetry`, `DefineTensor`, `UndefineTensorSymmetries`.

`Del` 4.1

`Del[x]` is the total covariant derivative of the tensor expression `x`. If `x` is a k-tensor, then `Del[x]` is a $(k+1)$-tensor. The last index slot of `Del[x]` is the one generated by differentiation; it is always covariant, and is associated with the underlying tangent bundle of the bundle(s) of `x`. If `x` is a scalar function (0-tensor), then `Del[x]` is automatically replaced by `Extd[x]`.

`Del[v,x]` is the covariant derivative of `x` in the direction `v`; `v` must be a 1-tensor expression. In output form, `Del[v,x]` appears as:

```
Del  [x]
   v
```

Covariant derivatives with respect to (contravariant) basis vectors are displayed without the name of the basis vector. For example, `Del[ Basis[L[i]], x ]` appears as

```
Del  [x]
   i
```

Options:

- `Connection -> cn` specifies that the covariant derivative is to be taken with respect to the connection `cn` instead of the default connection. Ordinarily, `cn` should be an expression of the form `Conn + diff`, where `diff` is a 3-tensor expression representing the "difference tensor" between the default connection and `cn`.

- `Metric -> g`: a symmetric 2-tensor expression, indicating that the covariant derivative is to be taken with respect to the Levi-Civita connection of `g` instead of the default metric for `x`'s bundle (which is assumed to be Riemannian).

See also `Grad`, `CovD`, `Extd`, `Div`, `Laplacian`, `CovDExpand`.

`Det` 7.3

If `x` is a 2-tensor expression, Ricci interprets `Det[x]` as the determinant of `x`, using the metric to convert `x` to a {`Contravariant,Covariant`} tensor if necessary.

`Det` is also a value for the global variable `$WedgeConvention`.

See also `Dot`, `$WedgeConvention`.

`Dimension`

`Dimension[bundle]` returns the dimension of `bundle`. See also `DefineBundle`.

`Div` 4.3

`Div[x]` is the divergence of the tensor expression `x`, which is the covariant derivative of `x` contracted on its last two indices. If `x` is a k-tensor, `Div[x]` is a $(k-1)$-tensor. Ricci assumes (without checking) that the last index of `x` is associated with the underlying tangent bundle of `x`'s bundle. `Div` is the formal adjoint of `-Del`.

Options:

- `Connection -> cn` specifies that the divergence is to be taken with respect to the connection `cn` instead of the default connection. Ordinarily, `cn` should be an expression of the form `Conn + diff`, where `diff` is a 3-tensor expression representing the "difference tensor" between the default connection and `cn`.
- `Metric -> g`: a symmetric 2-tensor expression, indicating that the divergence is to be taken with respect to the Levi-Civita connection of `g` instead of the default metric for `x`'s bundle (which is assumed to be Riemannian).

See also `Del`, `Laplacian`.

`DivGrad` 4.5

`DivGrad` is a value for the global variable `$LaplacianConvention`.

`Dot` 3.4, 7.3

When `x` and `y` are tensor expressions, `x.y` or `Dot[x,y]` represents the contraction of the last index of `x` with the first index of `y`. When `x` and `y` are 2-tensors, this can be interpreted as matrix multiplication. If `Dot` is applied to three or more tensor expressions, all arguments except possibly the first and last must have rank greater than 1. Ricci automatically expands dot products of sums and scalar multiples, and replaces dot products of 1-tensors with inner products. See also `Inner`, `Inverse`, `Kronecker`.

`ERROR`

If you attempt to insert the wrong number of indices into a tensor expression, Ricci returns `ERROR[expression]`.

`Even` 2.3

`Even` is a value for the `Type` option of `DefineConstant`.

`Expand` 5.3

If you apply the Mathematica function `Expand` to an expression that includes tensors with indices, Ricci automatically converts it to `TensorExpand`. See also `TensorExpand`.

`Extd` 4.6

`Extd[x]` is the exterior derivative of `x`, which must be an alternating covariant tensor expression. In output form, `Extd[x]` appears as `d[x]`. Ricci automatically expands exterior derivatives of sums, scalar multiples, powers, and wedge products. See also `Del`, `ExtdStar`, `LaplaceBeltrami`.

`ExtdStar` 4.7

`ExtdStar[x]` is the adjoint of the operator `Extd` applied to `x`, which must be an alternating covariant tensor expression. If `x` is a k-form, then `ExtdStar[x]` is a $(k-1)$-form. In output form, `ExtdStar[x]` appears as shown here:

```
In[26]:= ExtdStar[x]

                *
Out[26]= d [x]
```

Option:

- `Metric -> g`: a symmetric 2-tensor expression, representing a metric to be used in place of the default metric for `x`'s bundle (which is assumed to be Riemannian).

See also `Extd`, `LaplaceBeltrami`, `Div`.

`FactorConstants` 5.8

`FactorConstants[x]` applies the Mathematica function `Factor` to the constant factor in each term of the tensor expression `x`. `FactorConstants[x,n]` or `FactorConstants[x,{n1,n2,...}]` applies `FactorConstants` only to term `n` or to terms `n1,n2,...` of `x`. See also `CollectConstants`, `SimplifyConstants`, `ConstantFactor`, `TensorFactor`.

`FirstBianchiRule` 7.2

`FirstBianchiRule` is a rule that attempts to turn sums containing two 4-tensors with `Symmetries -> RiemannSymmetries` into a single term, using the first Bianchi identity. For example:

```
              i               i
Out[7]= 2 Rm          + 2 Rm
              j k l            l j k

In[8]:= % /. FirstBianchiRule

              i
Out[8]= 2 Rm
              k j l
```

See also `SecondBianchiRule`, `BianchiRules`, `ContractedBianchiRules`.

`FirstStructureRule` 7.5

`FirstStructureRule` is a rule that implements the first structure equation for exterior derivatives of basis covectors. For example (assuming `$WedgeConvention = Alt`):

```
                 i
Out[9]= d[Basis  ]

In[10]:= % /. FirstStructureRule

               k6           i    1    i
Out[10]= Basis    ^ Conn       + - Tor
                          k6      2
```

See also `SecondStructureRule`, `CompatibilityRule`, `StructureRules`, `$WedgeConvention`.

`FirstUp`

`FirstUp` is a value for the `RiemannConvention` option of `DefineBundle`.

`FlatConnection` 2.1

`FlatConnection` is a `DefineBundle` option.

The function `FlatConnection[bundle]` returns `True` or `False`.

See also `DefineBundle`.

`FormQ`

`FormQ[x]` returns `True` if `x` is a covariant alternating tensor expression, and `False` otherwise. See also `DefineTensor`.

`Grad` 4.4

`Grad[x]` is the gradient of the tensor expression `x`. It is the same as `Del[x]`, except that the last index is contravariant instead of covariant. If `x` is a scalar function (0-tensor), then `Grad[x]` is a vector field. See also `Del`, `Laplacian`.

`Hermitian` 2.1

`Hermitian` is a value for the `Symmetries` option of `DefineTensor`.

`HodgeInner` 3.6

`HodgeInner[x,y]` represents the Hodge inner product of the alternating tensors `x` and `y`, taken with respect to the default metric(s) on the bundle(s) with which `x` and `y` are associated. It is equal to the usual inner product multiplied by a numerical scale factor (depending on `$WedgeConvention`) chosen so that, if $\{e^i\}$ are orthonormal 1-forms, then the following k-forms are orthonormal:

$$e^{i_1} \wedge \ldots \wedge e^{i_k}, \quad i_1 < \ldots < i_k.$$

The arguments x and y must be alternating tensor expressions of the same rank. In output form, HodgeInner[x,y] appears as <<x, y>>. See also Inner, HodgeNorm, Int, ExtdStar.

HodgeNorm

HodgeNorm[x] is the norm of the alternating tensor expression x, with respect to the Hodge inner product. It is automatically converted by Ricci to Sqrt[HodgeInner[x,Conjugate[x]]]. See also HodgeInner, Norm.

Im 2.6

Ricci converts Im[x] to (x - Conjugate[x])/(2I). See also Conjugate.

Imaginary 2.3, 2.4

Imaginary is a value for the Type option of DefineConstant, DefineTensor, and DefineMathFunction.

IndexOrderedQ 6.1

IndexOrderedQ[{indices}] returns True if {indices} are ordered correctly according to Ricci's index ordering rules—first by name, then by altitude (lower before upper)—and False otherwise. See also OrderDummy, OrderCovD, DefineRelation, DefineRule.

IndexQ

IndexQ[i] returns True if i is an index name, and False otherwise. See also DefineBundle, DefineIndex.

Inner 3.5

If x and y are tensor expressions, Ricci interprets Inner[x,y] as the inner product of x and y, taken with respect to the default metric(s) on the bundle(s) with which x and y are associated. If x and y have opposite variance (for example, if x is a 1-form and y is a vector field), then Inner[x,y] represents the natural pairing between x and y. The arguments x and y must have the same rank. Ricci automatically expands inner products of sums and scalar multiples. In output form, Inner[x,y] appears as <x, y>. See also Norm, HodgeInner.

InsertIndices 2.6

If x is a tensor expression of rank k, then InsertIndices[x, {L[i1],...,L[ik]}] causes the indices L[i1], ..., L[ik] to be inserted into the k index slots in order, yielding a component expression. In input form, this can be abbreviated x[L[i1], ..., L[ik]]. Indices in each slot may be lower (L[i]) or upper (U[i]).

If `x` has rank 0, then `InsertIndices[ x, {} ]` or `x[]` converts it to a component expression by generating dummy indices as necessary.

If one or more indices are inserted into a rank-0 expression, then `InsertIndices` is automatically converted to `CovD`. See also `CovD`.

`Int` 3.7

If `x` and `y` are alternating tensor expressions with `Rank[x]` $\leq$ `Rank[y]`, `Int[x,y]` represents the generalized interior product of `x` into `y`. When `x` is a vector field, `Int[x,y]` is interior multiplication of `x` into `y` (with a numerical factor depending on `$WedgeConvention`). In general, `Int[x,_]` is the adjoint (with respect to the Hodge inner product) of wedging with `x` on the left: if `alpha`, `beta`, and `gamma` are alternating tensors of ranks $a, b, a+b$, respectively, then

```
<<alpha ^ beta, gamma>> = <<beta, Int     [gamma]>>.
                                     alpha
```

See also `HodgeInner`, `Wedge`, `Dot`, `$WedgeConvention`.

`Integer` 2.3

Ricci recognizes `Integer` as a value for the `Type` option of `DefineConstant`.

`Inverse` 7.3

If `x` is a 2-tensor expression, Ricci interprets `Inverse[x]` as the matrix inverse of `x`. See also `Dot`, `Kronecker`.

`Kronecker` 7.3

`Kronecker[L[i],U[j]]` is the Kronecker delta symbol, which is equal to 1 if `i = j` and 0 otherwise. `Kronecker` is a generic tensor representing the identity endomorphism of any bundle. See also `Dot`, `Inverse`.

`L` 2.2, 2.4

`L[i]` represents a lower index `i`. `L[i[Bar]]` is the internal representation for a lower barred index `i`; it can be abbreviated in input form by `LB[i]`. See also `LB`, `U`, `UB`.

`LaplaceBeltrami` 4.8

`LaplaceBeltrami[x]` is the Laplace-Beltrami operator $\Delta = dd^* + d^*d$ applied to the differential form `x`. It is automatically replaced by `Extd[ExtdStar[x]] + ExtdStar[Extd[x]]`.

Option:

- `Metric -> g`: a symmetric 2-tensor expression, representing a metric to be used in place of the default metric for `x`'s bundle (which is assumed to be Riemannian).

See also `Extd`, `ExtdStar`, `Laplacian`.

`Laplacian` 4.5

`Laplacian[x]` is the covariant Laplacian of the tensor expression `x`. The sign convention of `Laplacian` is controlled by the global variable `$LaplacianConvention`. When `$LaplacianConvention = DivGrad` (the default), `Laplacian[x]` is automatically replaced by `Div[Grad[x]]`. When `$LaplacianConvention = PositiveSpectrum`, `Laplacian[x]` is replaced by `-Div[Grad[x]]`.

Options:

- `Metric -> g`: a symmetric 2-tensor expression, representing a metric to be used in place of the default metric for the underlying tangent bundle of `x`.
- `Connection -> cn` specifies that covariant derivatives are to be taken with respect to the connection `cn` instead of the default connection. Ordinarily, `cn` should be an expression of the form `Conn + diff`, where `diff` is a 3-tensor expression representing the "difference tensor" between the default connection and `cn`.

See also `Div`, `Grad`, `LaplaceBeltrami`.

`LB` 2.2, 2.4

`LB[i]` is the input form for a lower barred index `i`. It is converted automatically to the internal form `L[i[Bar]]`. See also `L`, `U`, `UB`, `Bar`.

`LeviCivitaConnection` 7.6

`LeviCivitaConnection[g]` represents the Levi-Civita connection of the arbitrary Riemannian metric `g`. It behaves as a product tensor of rank {2,1}. When indices are inserted, the connection coefficients (Christoffel symbols) are computed in terms of the background connection of `g`'s bundle (assumed to be Riemannian) and covariant derivatives of `g`. The upper index is the *second* one, just as for the default connection `Conn`. See also `Conn`, `Curvature`, `RiemannTensor`, `RicciTensor`, `ScalarCurv`.

`Lie` 4.9

`Lie[v,x]` is the Lie derivative of the tensor expression `x` in the direction `v`. The first argument `v` must be a vector field (a contravariant 1-tensor expression). If `v` and `x` are both vector fields, `Lie[v,x]` is their Lie bracket, and Ricci automatically puts the factors in lexical order by inserting a minus sign if necessary. Applying `LieRule` causes Lie derivatives of differential forms to be expanded in terms of `Int` and exterior derivatives. See also `Del`, `LieRule`.

`LieRule` 4.9

`LieRule` is a rule that transforms Lie derivatives of differential forms to expressions involving exterior derivatives and `Int` according to the relation

```
Lie  [w] --> Int [d[w]] + d[Int[ [w]]]
   v            v                v
```

See also `Lie`, `Int`, `Extd`.

`LowerAllIndices` 5.20

`LowerAllIndices[x]` lowers all of the indices in `x` (except those appearing on metrics or `Basis` covectors) by inserting appropriate metrics with raised indices. `LowerAllIndices[x,n]` or `LowerAllIndices[x,{n1,n2,...}]` applies `LowerAllIndices` to term `n` or to terms `n1,n2,...` of `x`. See also `CorrectAllVariances`.

`Method`

`Method` is an option for the Ricci simplification command `OrderDummy`, which specifies how hard the command should work to simplify the expression.

`Metric`

`Metric -> metricname` is an option for `Del`, `Div`, `Grad`, `ExtdStar`, `CovD`, `Laplacian`, and `LaplaceBeltrami`.

The function `Metric[bundle]` returns the bundle's metric.

See also `DefineBundle`.

`MetricQ`

`MetricQ[x]` returns `True` if `x` is a metric with or without indices, and `False` otherwise. See also `DefineBundle`.

`MetricType` 7.2

`MetricType` is a `DefineBundle` option, which specifies whether the bundle's metric has special properties. Currently the only allowable values are `MetricType -> Normal` (no special properties), and `MetricType -> Riemannian` (for a Riemannian tangent bundle). The default is `Normal`.

`Negative` 2.3, 2.4, 2.6

Ricci recognizes `Negative` as a value for the `Type` option of `DefineConstant`, `DefineTensor`, and `DefineMathFunction`.

`NewDummy` 5.16

`NewDummy[x]` converts all dummy indices occurring in the tensor expression `x` to computer-generated dummy indices. `NewDummy[x,n]` or `NewDummy[x,{n1,n2,...}]` applies `NewDummy` to term `n` or to terms `n1,n2,...` of `x`. For most bundles, the generated dummy names are of the form `k`n, where `k` is the last index name associated with bundle and n is an integer. For one-dimensional bundles, only the dummy name `k` itself is generated. See also `RenameDummy`, `OrderDummy`.

`NewRule` 6.2

`NewRule` is a `DefineRule` option.

`NonNegative` 2.3, 2.4, 2.6

Ricci recognizes `NonNegative` as a value for the `Type` option of `DefineConstant`, `DefineTensor`, and `DefineMathFunction`.

`NonPositive` 2.3, 2.4, 2.6

`NonPositive` is a value for the `Type` option of `DefineConstant`, `DefineTensor`, and `DefineMathFunction`.

`NoOneDims`

`NoOneDims` is a value for the `Mode` option of `AbsorbMetrics`, used internally by `TensorSimplify`.

`Norm`

`Norm[x]` is the norm of the tensor expression `x`. It is automatically converted by Ricci to `Sqrt[Inner[x,Conjugate[x]]]`. See also `Inner`, `HodgeNorm`.

`Normal` 7.2

`Normal` is a value for the `MetricType` option of `DefineBundle`.

`NoSymmetries` 2.4

`NoSymmetries` is a value for the `Symmetries` option of `DefineTensor`.

`Odd` [47 2.3

`Odd` is a value for the `Type` option of `DefineConstant`.

`OneDims`

`OneDims` is a value for the `Mode` option of `CorrectAllVariances`, used internally by `TensorSimplify`.

`OrderCovD` 5.18

`OrderCovD[x]` puts all of the indices appearing after ";" in the tensor expression `x` in order according to Ricci's index-ordering rules (first alphabetically by name, then by altitude), by adding appropriate curvature and torsion terms. `OrderCovD[x,n]` or `OrderCovD[x,{n1,n2,...}]` applies `OrderCovD` only to term `n` or to terms `n1,n2,...` of `x`. See also `CommuteCovD`, `CovDSimplify`, `IndexOrderedQ`.

`OrderDummy` 5.6

`OrderDummy[x]` attempts to put the dummy indices occurring in the tensor expression `x` in a "canonical form". `OrderDummy[x,n]` or `OrderDummy[x,{n1,n2,...}]` applies `OrderDummy` to term `n` or to terms `n1,n2,...` of `x`. All pairs of dummy indices are ordered so that the lower member appears first whenever possible. Then `OrderDummy` tries various permutations of the dummy index names in each term of `x`, searching for the lexically smallest version of the expression among all equivalent versions.

Option:

- `Method -> n` specifies how hard `OrderDummy` should work to find the best possible version of the expression. The allowable values are 0, 1, and 2. The default is `Method -> 1`, which means that dummy index names are interchanged in pairs until the lexically smallest version of the expression is found; this is used by `TensorSimplify`. `Method -> 2` causes `OrderDummy` to try all possible permutations of the dummy index names; this is used by `SuperSimplify`. `Method -> 0` means don't try interchanging names at all. If n is the number of dummy index pairs in the expression, the time taken by `Method -> 1` is proportional to n^2, while the time taken by `Method -> 2` is proportional to $n!$. This can be very slow, especially if there are more than 4 dummy pairs per term.

See also `TensorSimplify`, `SuperSimplify`, `RenameDummy`, `NewDummy`.

`ParallelFrame` 2.1

`ParallelFrame` is a `DefineBundle` option.

`Plus` 2.6

Ricci transforms expressions of the form `(a + b)[L[i],...]` into `InsertIndices[ a+b, {L[i],...} ]`. Any number of upper and/or lower indices can be inserted in this way, provided that the number of indices is consistent with the rank of `a+b`.

`Positive` 2.3, 2.4, 2.6

Ricci recognizes `Positive` as a value for the `Type` option of `DefineConstant`, `DefineTensor`, and `DefineMathFunction`.

`PositiveDefinite` 2.1

`PositiveDefinite` is a `DefineBundle` option.

`PositiveSpectrum` 4.5

`PositiveSpectrum` is a value for the global variable `$LaplacianConvention`.

`Power` 2.6

If `x` is a tensor expression and `p` is a positive integer, Ricci interprets `x^p` as the p-th symmetric power of `x` with itself. Ricci transforms expressions of the form `(x^p)[i]` into `InsertIndices[x^p,{i}]` whenever `i` is an index (`L[j]` or `U[j]`).

Ricci causes a power of a product such as `(a b)^p` to be expanded into a product of powers `a^p * b^p`, provided `a` and `b` do not contain any indices; if they do contain indices, then `(a b)^p` is transformed to `Summation[a b]^p`, to prevent the expression from being expanded to `a^p b^p`. `Summation` is not printed in output form. See also `SymmetricProduct`, `Summation`, `PowerSimplify`, `ProductExpand`.

`PowerSimplify` 5.14

`PowerSimplify[x]` attempts to simplify powers that appear in the tensor expression `x`. `PowerSimplify[x,n]` or `PowerSimplify[x,{n1,n2,...}]` applies `PowerSimplify` only to term `n` or to terms `n1,n2,...` of `x`. `PowerSimplify` transforms `x` by

- Converting products like `a^p b^p` to `Summation[a b]^p` when `a` and `b` contain indices. `Summation` is an internal Ricci function that prevents such products from being automatically expanded by Mathematica. It does not appear in output form; instead, `Summation[a b]^p` prints as `(a b)^p`.
- Expanding powers like `(a^b)^c` to `a^(b c)`, provided it knows enough about `a`, `b`, and `c` to know this is legitimate (for example, if `c` is an integer, or if `a` is positive and `b` is real).
- Expanding and collecting constants in any expression that appears as base or exponent of a power.

See also `TensorSimplify`, `Power`, `Summation`.

`ProductExpand` 5.13

`ProductExpand[x]` expands symmetric products and wedge products of 1-tensors that occur in `x`, and rewrites them in terms of tensor products. `ProductExpand[x,n]` or `ProductExpand[x,{n1,n2,...}]` applies `ProductExpand` only to term `n` or to terms `n1,n2,...` of `x`. See also `BasisExpand`, `Sym`, `Alt`, `SymmetricProduct`, `Wedge`.

`Quiet`

`Quiet` is an option for some of the defining and undefining commands in the Ricci package. The option `Quiet -> True` silences the usual messages printed by these commands. The default is the value of the global variable `$Quiet`, which is initially `False`.

`Rank`

`Rank[x]` returns the rank of the tensor expression `x`. If `t` is a tensor without indices, `Rank[t]` is the total rank as specified in the call to `DefineTensor`. If `t` is a tensor with all index slots filled, `Rank[t]` $= 0$. If `t` is a product tensor with some, but not all, index slots filled, then `Rank[t]` is the total rank of `t` minus the number of filled slots. For any other tensor expression `x`, `Rank[x]` depends on the meaning of the expression. See also `DefineTensor`.

`Re` 2.6

Ricci converts `Re[x]` to `(x + Conjugate[x])/2`. See also `Conjugate`.

`Real` 2.1, 2.3, 2.4, 2.6

Ricci recognizes `Real` as a value for the `Type` option of `DefineBundle`, `DefineConstant`, `DefineTensor`, and `DefineMathFunction`.

`RenameDummy` 5.5

`RenameDummy[x]` changes the names of dummy indices in `x` to standard names. `RenameDummy[x,n]` or `RenameDummy[x, {n1,n2,...} ]` applies `RenameDummy` only to term `n` or to terms `n1,n2,...` of `x`. `RenameDummy` chooses dummy names in alphabetical order from the list of index names associated with the appropriate bundle, skipping those names that already appear in `x` as free indices. When the list of index names is exhausted, computer-generated names of the form `k`n are used, where `k` is the last index name in the list and n is an integer. For one-dimensional bundles, n is omitted. See also `TensorSimplify`, `OrderDummy`, `NewDummy`.

`RicciSave` 2.7

`RicciSave["filename"]` causes the definitions of all symbols defined in Mathematica's current context, along with Ricci's predefined tensors and global variables, to be saved in Mathematica input form into the file `filename`. (The current context is usually `Global‘`, and usually includes all symbols you have used in the current session.) The previous contents of `filename` are erased. You can read the definitions back in by typing `<<filename`.

`RicciTensor` 7.6

`RicciTensor -> name` is a `DefineBundle` option, which specifies a name to be used for the Ricci tensor for this bundle. The default is `Rc`. This option takes effect only when `MetricType -> Riemannian` has been specified.

`RicciTensor[g]` represents the Ricci curvature tensor of the arbitrary metric `g`. When indices are inserted, the components of the Ricci tensor are computed in terms of covariant derivatives of `g` and the curvature of the default metric on `g`'s bundle, which is assumed to be Riemannian. NOTE: If you insert any upper indices, they are considered to have been raised by the *default* metric on `g`'s bundle, not by `g`. To use `g` to raise indices, you must explicitly multiply by components of `Inverse[g]`.

See also `DefineBundle`, `RiemannTensor`, `ScalarCurv`, `LeviCivitaConnection`.

`RiemannConvention` 7.2

`RiemannConvention` is a `DefineBundle` option, which specifies the index convention of the Riemannian curvature tensor.

`Riemannian` 7.2

`MetricType -> Riemannian` is a `DefineBundle` option for defining a Riemannian tangent bundle.

`RiemannSymmetries`

`RiemannSymmetries` is a value for the `Symmetries` option of `DefineTensor`.

`RiemannTensor` 7.6

`RiemannTensor -> name` is a `DefineBundle` option, which specifies a name to be used for the Riemannian curvature tensor for this bundle. The default is `Rm`. This option takes effect only when `MetricType -> Riemannian` has been specified.

`RiemannTensor[g]` represents the Riemannian curvature tensor of the arbitrary metric `g`. Like the default Riemannian curvature tensor, `RiemannTensor[g]` is a covariant 4-tensor. When indices are inserted, the components of the curvature tensor are computed in terms of covariant derivatives of `g` and the curvature of the default metric on `g`'s bundle, which is assumed to be Riemannian. NOTE: If you insert any upper indices, they are considered to have been raised by the *default* metric on `g`'s bundle, not by `g`. To use `g` to raise indices, you must explicitly multiply by components of `Inverse[g]`.

See also `DefineBundle`, `LeviCivitaConnection`, `RicciTensor`, `ScalarCurv`.

`Same`

`Bundle -> Same` is a special `DefineTensor` option for product tensors.

`ScalarCurv` 7.6

`ScalarCurv -> name` is a `DefineBundle` option, which specifies a name to be used for the scalar curvature for this bundle. The default is `Sc`. This option takes effect only when `MetricType -> Riemannian` has been specified.

`ScalarCurv[g]` represents the scalar curvature of the arbitrary metric `g`. When indices are inserted, the scalar curvature is computed in terms of covariant derivatives of `g` and the curvature of the default metric on `g`'s bundle, which is assumed to be Riemannian.

See also `RiemannTensor`, `RicciTensor`, `LeviCivitaConnection`, `DefineBundle`.

`ScalarQ`

`ScalarQ[x]` is `True` if `x` is a rank 0 tensor expression with no free indices, and `False` otherwise.

`SecondBianchiRule` 7.2

`SecondBianchiRule` is a rule that attempts to turn sums containing two differentiated Riemannian curvature tensors into a single term using the second Bianchi identity. For example:

```
 2 Rm           + 2 Rm           /. SecondBianchiRule
     i j k l; m       i j l m; k
                                       --> -2 Rm
                                               i j m k; l
```

See also `FirstBianchiRule`, `ContractedBianchiRules`, `BianchiRules`, `DefineBundle`.

`SecondStructureRule` 7.5

`SecondStructureRule` is a rule that implements the second structure equation for exterior derivatives of the generic connection forms. For example:

```
Extd[Conn[L[i],U[j]]] /. SecondStructureRule -->

        1      j          k         j
        - Curv     + Conn    ^ Conn
        2    i          i          k
```

See also `FirstStructureRule`, `CompatibilityRule`, `StructureRules`, `Curv`, `Conn`, `CurvToConnRule`.

`SecondUp` 7.2

`SecondUp` is a value for the `RiemannConvention` option of `DefineBundle`.

`SimplifyConstants` 5.9

`SimplifyConstants[x]` applies the Mathematica function `Simplify` to the constant factor in each term of the tensor expression `x`. `SimplifyConstants[x,n]` or `SimplifyConstants[x,{n1,n2,...}]` applies `SimplifyConstants` only to term `n` or terms `n1,n2,...` of `x`. See also `CollectConstants`, `FactorConstants`, `ConstantFactor`.

`Skew` 2.4

`Skew` is a synonym for `Alternating`.

`SkewHermitian`

`SkewHermitian` is a value for the `Symmetries` option of `DefineTensor`.

`SkewQ`

`SkewQ[x]` is `True` if `x` is an alternating or skew-Hermitian tensor expression, and `False` otherwise.

`StructureRules` 7.5

`StructureRules` is the union of `CompatibilityRule`, `FirstStructureRule`, and `SecondStructureRule`.

`Summation`

If `a` and `b` contain indices, then Ricci transforms `(a b)^p` internally to `Summation[a b]^p`, to prevent the expression from being expanded to `a^p * b^p`. `Summation` is not printed in output form; instead, `Summation[a b]^p` appears as if it were `(a b)^p`. See also `Power`, `PowerSimplify`.

`SuperSimplify` 5.1, 5.2

`SuperSimplify[x]` attempts to put the tensor expression `x` into a canonical form, so that two expressions that are equal are usually identical after applying `SuperSimplify`. `SuperSimplify[x,n]` or `SuperSimplify[x,{n1,n2,...}]` applies `SuperSimplify` only to term `n` or to terms `n1`, `n2`, ...of `x`. `SuperSimplify` works exactly the same way as `TensorSimplify`, except that it calls `OrderDummy` with the option `Method -> 2`, so that all possible permutations of dummy index names are tried. This is generally much slower for expressions having more than 4 dummy index pairs: for an expression with k dummy index pairs per term, the time taken by `TensorSimplify` is proportional to k^2, while the time taken by `SuperSimplify` is proportional to $k!$. See also `TensorSimplify`, `OrderDummy`, `TensorCancel`.

`Sym` 3.9

`Sym[x]` represents the symmetrization of the tensor expression `x`. If `x` is symmetric, then `Sym[x] = x`. See also `Alt`, `SymmetricProduct`.

`Symmetric` 2.4

`Symmetric` is a value for the `Symmetries` option of `DefineTensor`.

`SymmetricProduct` 3.3

If `x`, `y`, and `z` are tensor expressions, `SymmetricProduct[x,y,z]` or `x * y * z` or `x y z` represents their symmetric product. Symmetric products are represented internally by ordinary multiplication. In output form, Ricci inserts explicit asterisks whenever non-scalar tensor expression are multiplied together, to remind the user that multiplication is being interpreted as symmetric product. Mathematically, the symmetric product is defined by $a*b = \mathrm{Sym}(a\otimes b)$. See also `Times`, `Power`, `Sym`, `ProductExpand`.

`SymmetricQ`

`SymmetricQ[x]` is `True` if `x` is a symmetric or Hermitian tensor expression, and `False` otherwise.

`Symmetries` 2.4

`Symmetries` is a `DefineTensor` option.

`TangentBundle` 2.1

`TangentBundle` is a `DefineBundle` option.

The function `TangentBundle[x]` returns the tangent bundle list for the bundle `x`. See also `UnderlyingTangentBundle`.

`Tensor` 2.4

Ricci's internal form for tensors is

```
Tensor[ name, {i,j,...}, {k,l,...} ]
```

where `name` is the tensor's name, `i,j,...` are the tensor indices (each of the form `L[i]` or `U[i]`), and `k,l,...` are the indices resulting from covariant differentiation. In input form, this can be typed `name [i,j,...] [k,l,...]`. Either set of indices in brackets can be omitted if it is empty. See also `DefineTensor`, `UndefineTensor`.

`TensorCancel` 5.21

`TensorCancel[x]` attempts to simplify each term of `x` by combining and cancelling common factors, even when the factors have different names for their dummy indices. `TensorCancel[x,n]` or `TensorCancel[x,{n1,n2,...}]` applies `TensorCancel` only to term `n` or to terms `n1,n2,...` of `x`. See also `TensorSimplify`, `SuperSimplify`.

`TensorData`

`TensorData[name]` is a list containing data for the tensor `name`, used internally by Ricci. See also `DefineTensor`.

`TensorExpand` 5.3

`TensorExpand[x]` expands products and positive integral powers in `x`, just as `Expand` does, but maintains correct dummy index conventions and does not expand constant factors. `TensorExpand[x,n]` or `TensorExpand[x,{n1,n2,...}]` applies `TensorExpand` to term `n` or to terms `n1,n2,...` of `x`. If you apply the Mathematica function `Expand` to an expression containing tensors with indices, it is automatically converted to `TensorExpand`. See also `TensorSimplify`, `Expand`, `BasisExpand`, `CovDExpand`, `ProductExpand`.

`TensorFactor`

`TensorFactor[x]` returns the product of all the non-constant factors in `x`, which should be a monomial. See also `ConstantFactor`.

`TensorMetricQ`

`TensorMetricQ[tensorname]` returns `True` if `tensorname` is the metric of some bundle, and `False` otherwise. See also `DefineBundle`.

`TensorProduct` 3.1

`TensorProduct[x,y,z]` or `TProd[x,y,z]` represents the tensor product of `x`, `y`, and `z`. Ricci automatically expands tensor products of sums and scalar multiples. In output form, tensor products appear as

```
x (X) y (X) z
```

See also `SymmetricProduct`, `Wedge`.

`TensorQ`

`TensorQ[name]` returns `True` if `name` is the name of a tensor, and `False` otherwise. See also `DefineTensor`.

`TensorRankList`

`TensorRankList` stores the list of ranks for a product tensor. Used internally by Ricci. See also `DefineTensor`.

`TensorSimplify` 5.1

`TensorSimplify[x]` attempts to put the tensor expression `x` into a canonical form, so that two expressions that are equal will usually be identical after applying `TensorSimplify`. `TensorSimplify[x,n]` or `TensorSimplify[x,{n1,n2,...}]` applies `TensorSimplify` to term `n` or to terms `n1,n2,...` of `x`.

`TensorSimplify` expands products and positive integer powers, simplifies powers, uses metrics to raise and lower indices, tries to rename all dummy indices in a canonical order, and collects all terms containing the same tensor factors but different constant factors. When there are two or more dummy index pairs associated with the same bundle, `TensorSimplify` tries exchanging dummy index names in pairwise, choosing the lexically smallest result. It does not reorder indices after the ";" (use `OrderCovD` or `CovDSimplify` to do that). `TensorSimplify` calls `CorrectAllVariances`, `TensorExpand`, `AbsorbMetrics`, `PowerSimplify`, `RenameDummy`, `OrderDummy`, and `CollectConstants`. See also `SuperSimplify`, `OrderCovD`, `CovDSimplify`, `TensorCancel`, `FactorConstants`, `SimplifyConstants`.

`TensorSymmetry`

A tensor symmetry is an object of the form

`TensorSymmetry[name,d,{perm1,sgn1,...,permN,sgnN}]`,

where `d` is a positive integer, `perm1,...,permN` are non-trivial permutations of $\{1, 2, \ldots, d\}$, and `sgn1,...,sgnN` are constants, usually ± 1. `TensorSymmetry` objects can be defined with `DefineTensorSymmetries` and used in the `DefineTensor` command. See also `DefineTensorSymmetries`, `DefineTensor`.

`TeXFormat`

`TeXFormat` is an option for `DefineTensor` and `DefineIndex`.

`Times` 2.6, 3.3

Ricci uses ordinary multiplication to represent symmetric products of tensors. Ricci transforms expressions of the form `(a * b)[L[i],...]` into `InsertIndices[ a*b, {L[i],...} ]`. Any number of upper and/or lower indices can be inserted in this way, provided that the number of indices is consistent with the rank of `a*b`. In output form, Ricci modifies Mathematica's usual ordering of factors: constants are printed first, then scalars, then higher-rank tensor expressions, and Ricci inserts explicit asterisks between tensor factors of rank greater than 0 to remind the user that multiplication is being interpreted as symmetric product. See also `SymmetricProduct`, `Power`, `ProductExpand`.

`Tor` 7.5

`Tor` is the name for the generic torsion tensor for the default connection in any bundle. It is a product tensor of rank {1,2}; inserting the first index yields the torsion 2-forms associated with the default basis. Inserting all three indices yields the components of the torsion tensor.

`TorsionFree` 2.1, 7.5

`TorsionFree` is a `DefineBundle` option.

The function `TorsionFree[bundle]` returns `True` or `False`.

See also `DefineBundle`.

`TotalRank`

`TotalRank[tensorname]` returns the total rank of the tensor `tensorname`. This function is used internally by Ricci. To compute the rank of an arbitrary tensor expression, use `Rank` instead. See also `Rank`, `DefineTensor`.

`TProd` 3.1

`TProd` is an abbreviation for `TensorProduct`.

`Tr` 7.3

> `Tr[x]` represents the trace of the 2-tensor expression `x`, computed with respect to the default metric on `x`'s bundle. See also `Dot`.

`Transpose` 7.3

> If `x` is a tensor expression, Ricci interprets `Transpose[x]` as `x` with its index positions reversed. See also `Dot`.

`Type` 2.1, 2.3, 2.4, 2.6

> `Type` is an option for `DefineBundle`, `DefineConstant`, `DefineTensor`, and `DefineMathFunction`.

`U` 2.2, 2.4

> `U[i]` represents an upper index `i`. `U[i[Bar]]` is the internal representation for an upper barred index `i`; it can be abbreviated in input form by `UB[i]`. See also `UB`, `L`, `LB`.

`UB` 2.2, 2.4

> `UB[i]` is the input form for an upper barred index `i`. It is converted automatically to the internal form `U[i[Bar]]`. See also `U`, `L`, `LB`, `Bar`.

`UndefineBundle` 2.1

> `UndefineBundle[bundle]` clears the definition of `bundle`, its metric, and all its indices. If `bundle` is a Riemannian tangent bundle, it also clears the definitions of its Riemann, Ricci, and scalar curvature tensors. If you try to perform computations with previous expressions involving `bundle`'s indices, you may get unpredictable results.
>
> Option:
>
> - `Quiet -> True` or `False`. Default is `False`, or the value of `$Quiet` if set.
>
> See also `DefineBundle`, `UndefineIndex`, `Declare`.

`UndefineConstant`

> `UndefineConstant[c]` removes `c`'s definition as a constant.
>
> Option:
>
> - `Quiet -> True` or `False`. Default is `False`, or the value of `$Quiet` if set.
>
> See also `DefineConstant`, `Declare`.

`UndefineIndex`

`UndefineIndex[i]` or `UndefineIndex[{i,j,k}]` removes the association of indices with their bundles. If you try to perform computations with previous expressions involving these indices, you may get unpredictable results.

Option:

- `Quiet -> True` or `False`. Default is `False`, or the value of `$Quiet` if set.

See also `DefineIndex`, `DefineBundle`, `UndefineBundle`, `Declare`.

`UndefineRelation` 6.1

`UndefineRelation[tensor]` deletes the relation previously defined for `tensor`. The argument must exactly match the first argument of the corresponding call to `DefineRelation`. NOTE: There is no `UndefineRule` function; to remove the definition of a rule defined by `DefineRule`, simply execute an assignment such as `rulename =.`, or call `DefineRule` with the option `NewRule -> True`.

Option:

- `Quiet -> True` or `False`. Default is `False`, or the value of `$Quiet` if set.

See also `DefineRelation`.

`UndefineTensor`

`UndefineTensor[tensorname]` clears the definition of `tensorname`. If you try to perform computations with previous expressions involving `tensorname`, you may get unpredictable results.

Option:

- `Quiet -> True` or `False`. Default is `False`, or the value of `$Quiet` if set.

See also `DefineTensor`, `Declare`.

`UndefineTensorSymmetries`

`UndefineTensorSymmetries[name]` deletes the `TensorSymmetry` object created by `DefineTensorSymmetries[name,...]`.

`UnderlyingTangentBundle`

`UnderlyingTangentBundle[x]` returns a list of bundles representing the underlying tangent bundle of the expression. The tangent bundle is the direct sum of the bundles in the list. It is assumed that all tensors used in a given expression have the same underlying tangent bundle.

`Variance` 2.4

`Variance` is a `DefineTensor` option.

The function `Variance[x]` returns a list of variances of the tensor expression `x`, one for each index slot.

See also `DefineTensor`, `Bundles`.

`VectorFieldQ`

`VectorFieldQ[x]` returns `True` if `x` is a contravariant 1-tensor expression, and `False` otherwise.

`Wedge` 3.2

`Wedge[x,y,z]` represents the wedge or exterior product of `x`, `y`, and `z`. The arguments of `Wedge` must be alternating tensor expressions. They need not be covariant tensors, however; Ricci handles wedge products of covariant, contravariant, and mixed tensors all together, using the metric to raise and lower indices as needed. Ricci automatically expands wedge products of sums and scalar multiples, and arranges factors in lexical order by inserting appropriate signs.

The interpretation of `Wedge` in terms of tensor products is determined by the global variable `$WedgeConvention`. In output form, wedge products are indicated with a caret:

```
x ^ y ^ z
```

See also `$WedgeConvention`, `Alt`, `TensorProduct`, `ProductExpand`.

8.2 Global variables

This section describes Ricci's *global variables*: these are symbols that can be set by the user to control the way Ricci evaluates certain expressions. For example, to change the value of `$WedgeConvention` to `Det`, just execute the assignment

```
$WedgeConvention = Det
```

The variables described below all begin with `$`, and are all saved automatically if you issue the `RicciSave` command.

`$DefaultTangentBundle` 2.1

The global variable `$DefaultTangentBundle` can be set by the user to a bundle or list of bundles. It is used by `DefineBundle` as the default value for the `TangentBundle` option, by `DefineTensor` as the default value for the `Bundle` option, and by `BasisExpand` and similar commands as the default bundle for tensors such as `Curv`, `Conn`, and `Tor` that are not intrinsically associated with any particular bundle. By default, `$DefaultTangentBundle` is set to the first bundle the user defines (or the direct sum of this bundle and its conjugate if the bundle is complex). If you want to override this default behavior, you can give a value to `$DefaultTangentBundle` before calling `DefineBundle`. For example:

```
$DefaultTangentBundle = {horizontal, vertical}
```

specifies that the tangent bundle of all subsequently-defined bundles is to be the direct sum of the bundles named `horizontal` and `vertical`. See also `DefineTensor`, `DefineBundle`, `BasisExpand`.

`$LaplacianConvention` 4.5

`$LaplacianConvention` determines which sign convention is used for the covariant Laplacian on functions and tensors.

`$LaplacianConvention = DivGrad` means that `Laplacian[x] = Div[Grad[x]]`, while `$LaplacianConvention = PositiveSpectrum` means that `Laplacian[x] = -Div[Grad[x]]`. Default is `DivGrad`. See also `Laplacian`.

`$MathFunctions`

`$MathFunctions` is a list of names that have been defined as scalar mathematical functions for use by Ricci. See also `DefineMathFunction`.

`$Quiet`

The global variable `$Quiet` is used by Ricci to determine whether the defining and undefining commands report on what they are doing. Setting `$Quiet=True` will silence these chatty commands. The default is `False`. It can be overridden for a particular command by specifying the `Quiet` option as part of the command call.

`$RiemannConvention` 7.2

The global variable `$RiemannConvention` can be set by the user to specify a default value for the `RiemannConvention` option of `DefineBundle`. The allowed values are `FirstUp` and `SecondUp` (the default). See also `DefineBundle`.

`$TensorFormatting` 2.2

The global variable `$TensorFormatting` can be set to `True` or `False` by the user to turn on or off Ricci's special output formatting of tensors and indices. Default is `True`.

`$TensorTeXFormatting`

The global variable `$TensorTeXFormatting` can be set to `True` or `False` by the user to turn on or off Ricci's special formatting of tensors in `TeXForm`. Default is `True`.

`$WedgeConvention` 3.2

The global variable `$WedgeConvention` can be set by the user to determine the interpretation of wedge products. The allowed values are `Alt` and `Det`. The default is `Alt`. Suppose α is a p-form and β a q-form:

`$WedgeConvention = Alt:`

$$\alpha \wedge \beta = \mathrm{Alt}(\alpha \otimes \beta) \quad \text{and} \quad Basis^1 \wedge \ldots \wedge Basis^n = \frac{1}{n!}\det$$

`$WedgeConvention = Det:`

$$\alpha \wedge \beta = \frac{(p+q)!}{p!q!}\mathrm{Alt}(\alpha \otimes \beta) \quad \text{and} \quad Basis^1 \wedge \ldots \wedge Basis^n = \det$$

See also `Wedge`, `Alt`.