AbsoluteD

- `AbsoluteD[`*tensor, t*`]` represents the absolute derivative of tensor expression with respect to t.
- `AbsoluteD[`*tensor,* {*u, v, ...*}`]` represents the absolute derivative with respect to the list of variables.

The absolute derivative is sometimes also known as the intrinsic derivative.

An absolute derivative given in representational form can later be expanded in terms of the metric and coordinate positions using `ExpandAbsoluteD`.

The output Format of the unexpanded absolute derivative can be changed with `SetDerivativeSymbols`.

See also: `ExpandAbsoluteD`, `SetDerivativeSymbols`, `CovariantD`, `PartialD`, `TotalD`.

Examples

```
In[1]:= Needs["TensorCalculus4`Tensorial`"]
```

Save the settings.

```
In[2]:= oldindices = CompleteBaseIndices;
        oldflavors = IndexFlavors;
        ClearIndexFlavor /@ oldflavors;
        DeclareIndexFlavor[{red, Red}]

In[6]:= DefineTensorShortcuts[{{S, T}, 1}, {{S}, 2}]
```

The absolute derivative of a tensor scalar is the total derivative of the scalar.

```
In[7]:= Tensor[F]
        AbsoluteD[%, t]
        % // FullForm

Out[7]= F

Out[8]= dF/dt

Out[9]//FullForm=
      TotalD[Tensor[F], t]
```

The absolute derivative of a tensor is designated with the following output form, using a capital D on the upper part of the derivative. Internally, it remains unevaluated until expanded with `ExpandAbsoluteD`.

```
In[10]:= Su[i]
         AbsoluteD[%, t]
         % // FullForm
```

Out[10]= S^i

Out[11]= $\frac{D\,S^i}{d\,t}$

```
Out[12]//FullForm=
AbsoluteD[Tensor[S, List[i], List[Void]], t]
```

Higher order derivatives can be calculated.

```
In[13]:= Sud[i, j]
         AbsoluteD[%, {u, v}]
```

Out[13]= $S^i{}_j$

Out[14]= $\frac{D^2\,S^i{}_j}{d\,u\,d\,v}$

Absolute derivatives are linear and follow Liebniz's rule. Nothing special is needed to handle flavored expressions.

```
In[15]:= a Su[i] + b Tu[j] // ToFlavor[red]
         AbsoluteD[%, t]
```

Out[15]= $a\,S^i + b\,T^j$

Out[16]= $a\,\frac{D\,S^i}{d\,t} + b\,\frac{D\,T^j}{d\,t}$

```
In[17]:= a Su[i] + b Tu[j] // ToFlavor[red]
         AbsoluteD[%, {u, v}]
```

Out[17]= $a\,S^i + b\,T^j$

Out[18]= $a\,\frac{D^2\,S^i}{d\,u\,d\,v} + b\,\frac{D^2\,T^j}{d\,u\,d\,v}$

```
In[19]:= Su[i] Tu[j] // ToFlavor[red]
         AbsoluteD[%, t]
```

Out[19]= $S^i\,T^j$

Out[20]= $\frac{D\,T^j}{d\,t}\,S^i + \frac{D\,S^i}{d\,t}\,T^j$

```
In[21]:= Su[i] Tu[j] // ToFlavor[red]
         AbsoluteD[%, {u, v}]
```

Out[21]= $S^i\,T^j$

Out[22]= $\frac{D\,S^i}{d\,v}\,\frac{D\,T^j}{d\,u} + \frac{D\,S^i}{d\,u}\,\frac{D\,T^j}{d\,v} + \frac{D^2\,T^j}{d\,u\,d\,v}\,S^i + \frac{D^2\,S^i}{d\,u\,d\,v}\,T^j$

The absolute derivative is more interesting when expanded.

Restore setting...

```
In[23]:= ClearTensorShortcuts[{{S, T}, 1}, {{S}, 2}]

In[24]:= DeclareBaseIndices @@ oldindices
         ClearIndexFlavor /@ IndexFlavors;
         DeclareIndexFlavor /@ oldflavors;
         Clear[oldindices, oldflavors]
```

AntiSymmetric

- `AntiSymmetric[`*indices, weighting : True*`][`*expr*`]` calculates the antisymmetric tensor expression associated with expr for the list of indices.
- `AntiSymmetric[{`*indices1*`}, {`*indicies2*`}..., `*weighting : True*`][`*expr*`]` uses multiple sets of symmetric indices.

With weighting True (the default value) the `p !` terms generated are divided by `p !` where `p` is the number of antisymmetric indices.

The indices operated on must be all up or all down.

The indices in the command must carry the flavor.

AntiSymmetric is automatically mapped over arrays, equations and sums.

See also: `Symmetric`, `TensorSymmetry`, `IndexChange`, `SymmetrizeSlots`, `SymmetrizePattern`.

Examples

```
In[1]:= Needs["TensorCalculus4`Tensorial`"]
```

Save old settings and declare a flavor...

```
In[2]:= oldflavors = IndexFlavors;
        ClearIndexFlavor /@ oldflavors;
        DeclareIndexFlavor[{red, Red}]

In[5]:= DefineTensorShortcuts[{e, 1}, {{T, S}, 2}, {T, 3}]

In[6]:= Tuu[i, j]
        % // AntiSymmetric[{i, j}]

Out[6]= T^{i j}

Out[7]= 1/2 (T^{i j} - T^{j i})
```

The indices in `AntiSymmetric` must carry the flavor.

```
In[8]:= Tuu[i, j] // ToFlavor[red]
        % // AntiSymmetric[red /@ {i, j}]

Out[8]= T^{i j}

Out[9]= 1/2 (T^{i j} - T^{j i})
```

AntiSymmetric will not work on mixed indices.

```
In[10]:= Tudd[i, j, k]
         % // AntiSymmetric[{i, j, k}]
```

$$\text{Out[10]=}\ T^{i}{}_{j\,k}$$

```
        AntiSymmetric::indices :
         AntiSymmetric indices {i, j, k} must be all be in the list
           of up, {i}, or all in the list of down, {j, k}, indices.
```

```
Out[11]= $Aborted
```

Antisymmetrization also works for higher order tensors.

```
In[12]:= Tddd[i, j, k] // ToFlavor[red]
         % // AntiSymmetric[red /@ {i, j, k}]
```

$$\text{Out[12]=}\ T_{i\,j\,k}$$

$$\text{Out[13]=}\ \frac{1}{6}\left(T_{i\,j\,k}-T_{i\,k\,j}-T_{j\,i\,k}+T_{j\,k\,i}+T_{k\,i\,j}-T_{k\,j\,i}\right)$$

```
In[14]:= Tdd[i, j] Sdd[m, n]
         % // AntiSymmetric[{i, m}, {j, n}]
```

$$\text{Out[14]=}\ S_{m\,n}\,T_{i\,j}$$

$$\text{Out[15]=}\ \frac{1}{2}\left(\frac{1}{2}S_{m\,n}\,T_{i\,j}-\frac{1}{2}S_{m\,j}\,T_{i\,n}\right)+\frac{1}{2}\left(-\frac{1}{2}S_{i\,n}\,T_{m\,j}+\frac{1}{2}S_{i\,j}\,T_{m\,n}\right)$$

In some cases, as in defining wedge products in exterior algebra and differential forms, the weighting factor is not wanted.

```
In[16]:= ed[i] ⊗ ed[j]
         Wedge[ed[i], ed[j]] == (% // AntiSymmetric[{i, j}, False])
```

$$\text{Out[16]=}\ e_i\otimes e_j$$

$$\text{Out[17]=}\ e_i\wedge e_j == e_i\otimes e_j-e_j\otimes e_i$$

Also, see the examples in Symmetric.

Restore the settings.

```
In[18]:= ClearTensorShortcuts[{T, 2}, {T, 3}]
```

```
In[19]:= ClearIndexFlavor /@ IndexFlavors;
         DeclareIndexFlavor /@ oldflavors;
         Clear[oldflavors]
```

ArrayExpansion

- `ArrayExpansion[`*i, j, ..., base : Automatic*`][expr]` will form an array on the indices i, j,..., in the expression. The range of the expansion is over the base list, which has the default value of the base indices associated with the index flavor.
- `ArrayExpansion[{`*i, j, ...*`}, base : Automatic][expr]` may also be used.

The expansion indices must carry their flavors. base indices are not flavored.

The first index will be at the highest level in the resulting array, and the last index at the lowest level.

If special sets of base indices have been associated with certain flavors of indices using `DeclareBaseIndices`, then those sets will be used with the corresponding flavors.

The optional argument base gives the base indices over which each index is expanded. The default value is `Automatic` and then each index is expanded over the complete set of base indices for the corresponding flavor. If a list of subsets of selected base indices is given then each index is expanded over the corresponding selected subset taken in corresponding order. If a single list of selected base indices is supplied, then it will apply only to the first index.

`EinsteinArray`, which automatically expands on free indices, will often be more convenient to use.

See also: `DeclareBaseIndices`, `SumExpansion`, `EinsteinSum`, `EinsteinArray`, `ToArrayValues`.

Examples

```
In[1]:= Needs["TensorCalculus4`Tensorial`"]
```

Save the settings and declare base indices and flavors.

```
In[2]:= oldindices = CompleteBaseIndices;
        oldflavors = IndexFlavors;
        ClearIndexFlavor /@ oldflavors;
        DeclareBaseIndices[{1, 2, 3}]
        DeclareIndexFlavor /@ {{red, Red}, {rocket, SuperStar}};

In[7]:= DefineTensorShortcuts[{{x, y}, 1}, {S, 2}]
```

The following expands vectors into their components...

```
In[8]:= xu[i]
        % // ArrayExpansion[i]

Out[8]= x^i

Out[9]= {x^1, x^2, x^3}

In[10]:= xu[i] + yu[i]
         % // ArrayExpansion[i]

Out[10]= x^i + y^i

Out[11]= {x^1 + y^1, x^2 + y^2, x^3 + y^3}
```

An expansion can be done over a subset of the BaseIndices.

```
In[12]:= xu[i] + yu[i]
         % // ArrayExpansion[i, {1, 3}]
```

Out[12]= $x^i + y^i$

Out[13]= $\{x^1 + y^1, x^3 + y^3\}$

With different indices we obtain a two-dimensional array. (However, this is not a proper tensor expression.)

```
In[14]:= xu[i] + yu[j]
         % // ArrayExpansion[i, j] // MatrixForm
```

Out[14]= $x^i + y^j$

Out[15]//MatrixForm=

$$\begin{pmatrix} x^1 + y^1 & x^1 + y^2 & x^1 + y^3 \\ x^2 + y^1 & x^2 + y^2 & x^2 + y^3 \\ x^3 + y^1 & x^3 + y^2 & x^3 + y^3 \end{pmatrix}$$

The following expands a second order tensor into an array. The expansion indices must match the flavor of the expression indices.

```
In[16]:= Suu[i, j] // ToFlavor[red]
         % // ArrayExpansion[red@i, red@j] // MatrixForm
```

Out[16]= $S^{i\,j}$

Out[17]//MatrixForm=

$$\begin{pmatrix} S^{1\,1} & S^{1\,2} & S^{1\,3} \\ S^{2\,1} & S^{2\,2} & S^{2\,3} \\ S^{3\,1} & S^{3\,2} & S^{3\,3} \end{pmatrix}$$

The order of the expansion follows the order of the indices so this produces the transpose of the above matrix.

```
In[18]:= Suu[i, j] // ToFlavor[red]
         % // ArrayExpansion[red@j, red@i] // MatrixForm
```

Out[18]= $S^{i\,j}$

Out[19]//MatrixForm=

$$\begin{pmatrix} S^{1\,1} & S^{2\,1} & S^{3\,1} \\ S^{1\,2} & S^{2\,2} & S^{3\,2} \\ S^{1\,3} & S^{2\,3} & S^{3\,3} \end{pmatrix}$$

We can expand on subsets of the array by specifying the range for each index in the optional baseindices argument.

```
In[20]:= Suu[i, j] // ToFlavor[red]
         % // ArrayExpansion[red@i, red@j, {{1, 2}, {1, 2, 3}}] // MatrixForm
```

Out[20]= $S^{i\,j}$

Out[21]//MatrixForm=

$$\begin{pmatrix} S^{1\,1} & S^{1\,2} & S^{1\,3} \\ S^{2\,1} & S^{2\,2} & S^{2\,3} \end{pmatrix}$$

A single list will only be applied to the first index.

```
In[22]:= Suu[i, j] // ToFlavor[red]
         % // ArrayExpansion[red@i, red@j, {1, 3}] // MatrixForm
```

```
Out[22]= S^{i j}
```

```
Out[23]//MatrixForm=
```

$$\begin{pmatrix} S^{1\,1} & S^{1\,2} & S^{1\,3} \\ S^{3\,1} & S^{3\,2} & S^{3\,3} \end{pmatrix}$$

The following uses the same subset of base indices for each index expansion.

```
In[24]:= Suu[i, j] // ToFlavor[red]
         % // ArrayExpansion[red@i, red@j, Table[{1, 3}, {2}]] // MatrixForm
```

```
Out[24]= S^{i j}
```

```
Out[25]//MatrixForm=
```

$$\begin{pmatrix} S^{1\,1} & S^{1\,3} \\ S^{3\,1} & S^{3\,3} \end{pmatrix}$$

If the subset is not contained in the base indices an error occurs.

```
In[26]:= Suu[i, j] // ToFlavor[red]
         % // ArrayExpansion[red@i, red@j, {1, 4}] // MatrixForm
```

```
Out[26]= S^{i j}
```

```
SumArrayExpansion::subset : {1, 4} is not a subset of the base indices {1, 2, 3}
```

```
Out[27]= $Aborted
```

The following expands a set of equations.

```
In[28]:= yu[i] == Sud[i, j] xu[j] // ToFlavor[rocket]
         % // SumExpansion[rocket@j]
         % // ArrayExpansion[rocket@i] // TableForm
```

```
Out[28]= y^{i*} == S^{i*}_{j*} x^{j*}
```

```
Out[29]= y^{i*} == S^{i*}_{1*} x^{1*} + S^{i*}_{2*} x^{2*} + S^{i*}_{3*} x^{3*}
```

```
Out[30]//TableForm=
y^{1*} == S^{1*}_{1*} x^{1*} + S^{1*}_{2*} x^{2*} + S^{1*}_{3*} x^{3*}
y^{2*} == S^{2*}_{1*} x^{1*} + S^{2*}_{2*} x^{2*} + S^{2*}_{3*} x^{3*}
y^{3*} == S^{3*}_{1*} x^{1*} + S^{3*}_{2*} x^{2*} + S^{3*}_{3*} x^{3*}
```

If we have declared special base indices for some flavors of indices, then they are expanded on the corresponding bases.

```
In[31]:= DeclareBaseIndices[{1, 2, 3}, {red, {A, B}}]
```

```
In[32]:= Suu[i, red@j]
         % // ArrayExpansion[i, red@j] // MatrixForm

Out[32]= S^i j
```

Out[33]//MatrixForm=

$$\begin{pmatrix} S^{1\,A} & S^{1\,B} \\ S^{2\,A} & S^{2\,B} \\ S^{3\,A} & S^{3\,B} \end{pmatrix}$$

We can still use selected subsets for each index but, of course, they must be from the corresponding base sets.

```
In[34]:= Suu[i, red@j]
         % // ArrayExpansion[i, red@j, {{1, 2}, {B}}] // MatrixForm

Out[34]= S^i j
```

Out[35]//MatrixForm=

$$\begin{pmatrix} S^{1\,B} \\ S^{2\,B} \end{pmatrix}$$

```
In[36]:= DeclareBaseIndices[{1, 2, 3}, {red, {1, 2, 3}}, {rocket, {1, 2}}]
         Suu[red@i, rocket@j]
         % // ArrayExpansion[red@i, rocket@j] // MatrixForm

Out[37]= S^i j*
```

Out[38]//MatrixForm=

$$\begin{pmatrix} S^{1\,1*} & S^{1\,2*} \\ S^{2\,1*} & S^{2\,2*} \\ S^{3\,1*} & S^{3\,2*} \end{pmatrix}$$

Restore the initial values...

```
In[39]:= ClearTensorShortcuts[{{x, y}, 1}, {S, 2}]

In[40]:= DeclareBaseIndices @@ oldindices
         ClearIndexFlavor /@ IndexFlavors;
         DeclareIndexFlavor /@ oldflavors;
         Clear[oldindices, oldflavors]
```

BaseIndexQ

■ `BaseIndexQ[index]` returns True if index has a base index value as its raw index and False otherwise.

`{i, red[i], rocket[i]}` all have the raw index i.

`BaseIndexQ` takes into account any special flavors used in `DeclareBaseIndices`.

See also: `DeclareBaseIndices`, `BaseIndices`, `NDim`, `DeclareBaseIndices`.

Examples

```
In[1]:= Needs["TensorCalculus4`Tensorial`"]
```

```
In[2]:= oldindices = CompleteBaseIndices;
        oldflavors = IndexFlavors;
        ClearIndexFlavor /@ oldflavors;
```

This sets the linear space and index flavors.

```
In[5]:= DeclareBaseIndices[Range[3]]
        IndexFlavors = {};
        DeclareIndexFlavor /@ {{red, Red}, {blue, Blue}, {rocket, OverTilde}};
        {NDim, BaseIndices}
```

```
Out[8]= {3, {1, 2, 3}}
```

These should all be True...

```
In[9]:= {1, 2, 3, red[1], red[2], red[3], blue[1],
         blue[2], blue[3], rocket[1], rocket[2], rocket[3]}
        BaseIndexQ /@ %
```

```
Out[9]= {1, 2, 3, 1, 2, 3, 1, 2, 3, 1̃, 2̃, 3̃}
```

```
Out[10]= {True, True, True, True, True, True, True, True, True, True, True, True}
```

These should all be False...

```
In[11]:= {4, red[4], blue[4], rocket[4], i, j, k}
         BaseIndexQ /@ %
```

```
Out[11]= {4, 4, 4, 4̃, i, j, k}
```

```
Out[12]= {False, False, False, False, False, False, False}
```

The following gives a warning message because green is not a current flavor.

```
In[13]:= BaseIndexQ[green@1]

       RawIndex::notindex : green[1] is not a Symbol, Integer or Flavor.

Out[13]= False
```

In the following we declare a different set of base indices for red flavored indices.

```
In[14]:= DeclareBaseIndices[{1, 2, 3}, {red, {A, B}}]
```

These should all be True...

```
In[15]:= {1, 2, 3, red@A, red@B}
         BaseIndexQ /@ %

Out[15]= {1, 2, 3, A, B}

Out[16]= {True, True, True, True, True}
```

These should all be False.

```
In[17]:= {0, 4, A, B, red@1, red@2, red@C}
         BaseIndexQ /@ %

Out[17]= {0, 4, A, B, 1, 2, C}

Out[18]= {False, False, False, False, False, False, False}
```

Restore the initial state.

```
In[19]:= DeclareBaseIndices @@ oldindices
         ClearIndexFlavor /@ IndexFlavors;
         DeclareIndexFlavor /@ oldflavors;
         Clear[oldindices, oldflavors]
```

BaseIndices

■ `BaseIndices` gives the current set of base indices for the underlying linear space.

BaseIndices are set by the first argument in `DeclareBaseIndices`. They determine the base index set for all indices except those that have been assigned special base sets in the `DeclareBaseIndices` statement.

The length of `BaseIndices` determines `NDim`, the dimension of the underlying linear space.

Base indices may be integers or symbols. You could use `{0, 1, 2, 3}` for relativity, or `{t, x, y, z}`, or `{ρ, Θ, ϕ}` for a spherical coordinate system.

On loading, Tensorial automatically initializes the base indices to `{1, 2, 3}`.

Base indices are not in themselves flavored, but flavors may be added to them.

See also: `DeclareBaseIndices`, `CompleteBaseIndices`, `NDim`, `BaseIndexQ`.

Examples

```
In[1]:= Needs["TensorCalculus4`Tensorial`"]
```

The following gives the current set of base indices...

```
In[2]:= oldindices = CompleteBaseIndices

Out[2]= {{1, 2, 3}}
```

The base indices can be changed with

```
In[3]:= DeclareBaseIndices[{t, x, y, z}]
        BaseIndices

Out[4]= {t, x, y, z}
```

This resets to the old indices.

```
In[5]:= DeclareBaseIndices @@ oldindices
        BaseIndices
        Clear[oldindices]

Out[6]= {1, 2, 3}
```

BasisDotProductRules

- `BasisDotProductRules[e, g]` generates the rules that convert dot products of basis vectors to metric tensor components, using e as the base vector label and `g` as the metric label..

The routine is used by EvaluateSlots.

See also: `EvaluateSlots`, `ReverseBasisDotProductRules`.

Examples

```
In[1]:= Needs["TensorCalculus4`Tensorial`"]
        DefineTensorShortcuts[{{u, v, e}, 1}, {g, 2}]
```

We can just look at the rules produced. (Developer`ContextFreeForm worked in *Mathematica* Version 5.0.1 but no longer works in later versions.)

```
In[3]:= BasisDotProductRules[e, g] // Developer`ContextFreeForm
```

```
Out[3]= {e^TensorCalculus4`Tensorial`Private`i$_ . e^TensorCalculus4`Tensorial`Private`j$_ →
           g^TensorCalculus4`Tensorial`Private`i$TensorCalculus4`Tensorial`Private`j$,
         e_TensorCalculus4`Tensorial`Private`i$_ . e_TensorCalculus4`Tensorial`Private`j$_ →
           g_TensorCalculus4`Tensorial`Private`i$TensorCalculus4`Tensorial`Private`j$,
         e^TensorCalculus4`Tensorial`Private`i$_ . e_TensorCalculus4`Tensorial`Private`j$_ →
           g^TensorCalculus4`Tensorial`Private`i$
             TensorCalculus4`Tensorial`Private`j$,
         e_TensorCalculus4`Tensorial`Private`i$_ . e^TensorCalculus4`Tensorial`Private`j$_ →
           g_TensorCalculus4`Tensorial`Private`i$
             TensorCalculus4`Tensorial`Private`j$}
```

The following seems to be the current method of eliminated contexts but only if it's printed. And it may not work in the future.

```
In[4]:= Block[{Internal`$ContextMarks = False},
         Print[BasisDotProductRules[e, g]]]
```

```
{e^i$_ . e^j$_ → g^i$ j$, e_i$_ . e_j$_ → g_i$ j$, e^i$_ . e_j$_ → g^i$ _j$, e_i$_ . e^j$_ → g_i$ ^j$}
```

Here is an example of its use.

```
In[5]:= Print["Dot product of two vectors"]
        u.v
        Print["Substituting the component expressions"]
        %% /. {u → uu[i] ed[i], v → vu[j] ed[j]}
        Print["Using the linearity of dot products on basis vectors"]
        %% // LinearBreakout[Dot][eu[_], ed[_]]
        Print["Using BasisDotProductRules to convert to metric tensor components"]
        %% /. BasisDotProductRules[e, g]
        Print["Expanding after lowering an index with the metric"]
        %% // MetricSimplify[g]
        % // EinsteinSum[]
```

Dot product of two vectors

Out[6]= $u.v$

Substituting the component expressions

Out[8]= $(e_i\, u^i).\left(e_j\, v^j\right)$

Using the linearity of dot products on basis vectors

Out[10]= $e_i.e_j\, u^i\, v^j$

Using BasisDotProductRules to convert to metric tensor components

Out[12]= $g_{i\,j}\, u^i\, v^j$

Expanding after lowering an index with the metric

Out[14]= $u_j\, v^j$

Out[15]= $u_1\, v^1 + u_2\, v^2 + u_3\, v^3$

```
In[16]:= ClearTensorShortcuts[{{u, v, g}, 1}, {g, 2}]
```

BasisExpandCovariantD

- `BasisExpandCovariantD[{x, δ, g, Γ}, a, permissive : False][expr]` will expand first order covariant derivatives of tensors using x as the label for the coordinate positions, δ as the label for the Kronecker, g as the label for the metric tensor and Γ as the lavel for Christoffel symbols. The introduced dummy index will be a.
- `BasisExpandCovariantD[{x, δ, g, Γ}, {a, b, ...}, permissive : False][expr]` expands higher order covariant derivatives using the list of dummy indices.

BasisExpandCovariantD works just like ExpandCovariantD except that it allows expansion on expressions that contain base indices.

See also: `ExpandCovariantD`, `CovariantD`, `SetChristoffelValueRules`, `PartialD`, `AbsoluteD`, `TotalD`, `Tensor`.

Examples

```
In[1]:= Needs["TensorCalculus4`Tensorial`"]
```

Save the old settings.

```
In[2]:= oldindices = CompleteBaseIndices;
```

Define standard labels and shortcuts.

```
In[3]:= DeclareBaseIndices[{1, 2, 3}]
        labs = {x, δ, g, Γ};
        DefineTensorShortcuts[{T, 2}]
```

We can always expand a covariant derivative expression and then substitute specific base indices.

```
In[6]:= CovariantD[Tuu[i, j], k]
        % // ExpandCovariantD[labs, a]
        % // ArrayExpansion[i, {1}] // First
```

```
Out[6]= T^{i j}_{;k}
```

$$\text{Out[7]=}\quad T^{a\,j}\,\Gamma^{i}{}_{k\,a} + T^{i\,a}\,\Gamma^{j}{}_{k\,a} + \frac{\partial T^{i\,j}}{\partial x^{k}}$$

$$\text{Out[8]=}\quad T^{a\,j}\,\Gamma^{1}{}_{k\,a} + T^{1\,a}\,\Gamma^{j}{}_{k\,a} + \frac{\partial T^{1\,j}}{\partial x^{k}}$$

With `BasisExpandCovariantD` we can generate component expressions directly.

```
In[9]:= CovariantD[Tuu[1, j], k]
        % // BasisExpandCovariantD[labs, a]
```

```
Out[9]= T^{1 j}_{;k}
```

$$\text{Out[10]=}\quad T^{a\,j}\,\Gamma^{1}{}_{k\,a} + T^{1\,a}\,\Gamma^{j}{}_{k\,a} + \frac{\partial T^{1\,j}}{\partial x^{k}}$$

Restore settings.

```
In[11]:= ClearTensorShortcuts[{T, 2}]

In[12]:= DeclareBaseIndices @@ oldindices
         Clear[oldindices]
```