16.11 CANTENS: A Package for Manipulations and Simplifications of Indexed Objects

This package creates an environment which allows the user to manipulate and simplify expressions containing various indexed objects like tensors, spinors, fields and quantum fields.

Author: Hubert Caprasse.

16.11.1 Introduction

`CANTENS` is a package that creates an environment inside REDUCE which allows the user to manipulate and simplify expressions containing various indexed objects like tensors, spinors, fields and quantum fields. Briefly said, it allows him

- to define generic indexed quantities which can eventually depend implicitly or explicitly on any number of variables;
- to define one or several affine or metric (sub-)spaces, and to work within them without difficulty;
- to handle dummy indices and simplify adequatly expressions which contain them.

Beside the above features, it offers the user:

1. Several invariant elementary tensors which are always used in the applications involving the use of indexed objects like `delta, epsilon, eta` and the generalized delta function.
2. The possibility to define any metric and to make it bloc-diagonal if he wishes to.
3. The capability to symmetrize or antisymmetrize any expression.
4. The possibility to introduce any kind of symmetry (even partial symmetries) for the indexed objects.
5. The choice to work with commutative, non-commutative or anticommutative indexed objects.

In this package, one cannot find algorithms or even specific objects (i.e. like the covariant derivative or the SU(3) group structure constants) which are of used either in nuclear and particle physics. The objective of the package is simply to allow the user to easily formulate *his algorithms* in the *notations he likes most*. The package is also conceived so as to minimize the number of new commands. However, the large number of new capabilities inherently implies that quite a substantial number of new functions and commands must be used. On the other hand, in order to avoid too many error or warning messages the package assumes, in many cases, that the user is reponsible of the consistency of its inputs. The author is aware that the package is still perfectible and he will be grateful to all people who shall spare some time to communicate bugs or suggest improvements.

The documentation below is separated into four sections. In the first one, the space(s) properties and definitions are described.

In the second one, the commands to geberate and handle generic indexed quantities (called abusively tensors) are illustrated. The manipulation and control of free and dummy indices is discussed.

In the third one, the special tensors are introduced and their properties discussed especially with respect to their ability to work simultaneously within several subspaces.

The last section, which is also the most important, is devoted entirely to the simplification function `CANONICAL`. This function originates from the package `DUMMY` and has been substantially extended . It takes account of all symmetries, make dummy summations and seeks a "canonical" form for any tensorial expression. Without it, the present package would be much less useful.

Finally, an **index** has been created. It contains numerous references to the text. Different typings have been adopted to make a clear distinction between them. The conventions are the following:

- Procedure keywords are typed in capital roman letters.
- Package keywords are typed in typewriter capital letters.
- Cantens package keywords are in small typewriter letters.
- All other keywords are typed in small roman letters.

When `CANTENS` is loaded, the packages `ASSIST` and `DUMMY` are also loaded.

16.11.2 Handling of space(s)

One can work either in a *single* space environment or in a multiple space environment. After the package is loaded, the single space environment is set and a unique space is defined. It is euclidian, and has a symbolic dimension equal to `dim`. The single space environment is determined by the switch `ONESPACE` which is turned on. One can verify the above assertions as follows :

```
onespace ?; => yes

wholespace_dim ?; => dim

signature ?; => 0
```

One can introduce a pseudoeuclidian metric for the above space by the command `SIGNATURE` and verify that the signature is indeed 1:

```
signature 1;

signature ?; => 1
```

In principle the signature may be set to any positive integer. However, presently, the package cannot handle signatures larger than 1. One gets the Minkowski-like space metric

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

which corresponds to the convention of high energy physicists. It is possible to change it into the astrophysicists convention using the command `GLOBAL_SIGN`:

```
global_sign ?; => 1

global_sign (-1);

global_sign ?; => -1
```

This means that the actual metric is now $(-1, 1, 1, 1)$. The space dimension may, of course, be assigned at will using the function `WHOLESPACE_DIM`. Below, it is assigned to 4:

```
wholespace_dim 4; ==> 4
```

When the switch `ONESPACE` is turned off, the system *assumes* that this default space is non-existent and, therefore, that the user is going to define the space(s) in which he wants to work. Unexpected error messages will occur if it is not done. Once the switch is turned off many more functions become active. A few of them are available in the algebraic mode to allow the user to properly conctruct and control the properties of the various (sub-)spaces he is going to define and, also, to assign symbolic indices to some of them.

`DEFINE_SPACES` is the space constructor and `wholespace` is a reserved identifier which is meant to be the name of the global space if subspaces are introduced. Suppose we want to define a unique space, we can choose for its any name but choosing `wholespace` will be more efficient. On the other hand, it leaves open the possibility to introduce subspaces in a more transparent way. So one writes, for instance,:

```
define_spaces wholespace=

        {6,signature=1,indexrange=0 .. 5}; ==>t
```

The arguments inside the list, assign respectively the dimension, the signature and the range of the numeric indices which is allowed. Notice that the range starts from 0 and not from 1. This is made to conform with the usual convention for spaces of signature equal to 1. However, this is not compulsory. Notice that the declaration of the indexrange may be omitted if this is the only defined space. There are two other options which may replace the signature option, namely `euclidian` and `affine`. They have both an obvious significance.

In the subsequent example, an eleven dimension global space is defined and two subspaces of this space are specified. Notice that no indexrange has been declared for the entire space. However, the indexrange declaration is compulsory for subspaces otherwise the package will improperly work when dealing with numeric indices.

```
define_spaces wholespace={11,signature=1}; ==> t

define_spaces mink=

          {4,signature=1,indexrange=0 .. 3}; ==> t

define_spaces eucl=

          {6,euclidian,indexrange=4 .. 9}; ==> t
```

To remind ones the space context in which one is working, the use of the function `SHOW_SPACES` is required. Its output is an *algebraic value* from which the user can retrieve all the informations displayed. After the declarations above, this function gives:

```
show_spaces(); ==>

          {{wholespace,11,signature=1}

           {mink,4,signature=1,indexrange=0..3},
                 {eucl,6,euclidian,indexrange=4..9}}
```

If an input error is made or if one wants to change the space framework, one cannot directly redefine the relevant space(s). For instance, the input

```
define_spaces eucl=

          {7,euclidian,indexrange=4 .. 9}; ==>

    *** Warning: eucl cannot be (or is already)
                 defined as space identifier
       t
```

whih aims to fill all dimensions present in `wholespace` tells that the space `eucl` cannot be redefined. To redefine it effectively, one is to *remove* the existing definition first using the function `REM_SPACES` which takes any number of space-names as its argument. Here is the illustration:

```
rem_spaces eucl; ==> t

show_spaces(); ==>

           {{wholespace,11,signature=1},

             {mink,4,signature=1,indexrange=0..3}}

define_spaces eucl=

           {7,euclidian,indexrange=4 .. 10}; ==> t

show_spaces(); ==>

            {{wholespace,11,signature=1},

              {mink,4,signature=1,indexrange=0..3},

              {eucl,7,euclidian,indexrange=4..10}}
```

Here, the user is entirely responsible of the coherence of his construction. The system does NOT verify it but will incorrectly run if there is a mistake at this level.

When two spaces are direct product of each other (as the color and Minkowski spaces in quantum chromodynamics), it is not necessary to introduce the global space `wholespace`.

"Tensors" and symbolic indices can be declared to belong to a specific space or subspace. It is in fact an essential ingredient of the package and make it able to handle expressions which involve quantities belonging to several (sub-)spaces or to handle bloc-diagonal "tensors". This will be discussed in the next section. Here, we just mention how to declare that some set of symbolic indices belong to a specific (sub-)space or how to declare them to belong to any space. The relevant command is `MK_IDS_BELONG_SPACE` whose syntax is

```
    mk_ids_belong_space(<list of indices>,
                        <space | subspace identifier>)
```

For example, within the above declared spaces one could write:

```
    mk_ids_belong_space({a0,a1,a2,a3},mink); ==> t

    mk_ids_belong_space({x,y,z,u,v},eucl); ==> t
```

The command `MK_IDS_BELONG_ANYSPACE` allows to remake them usable either in `wholespace` if it is defined or in anyone among the defined spaces. For instance, the declaration:

```
 mk_ids_belong_anyspace a1,a2; ==> t
```

tells that a1 and a2 belong either to `mink` or to `eucl` or to `wholespace`.

16.11.3 Generic tensors and their manipulation

Definition

The generic tensors handled by `CANTENS` are objects much more general than usual tensors. The reason is that they are not supposed to obey well defined transformation properties under a change of coordinates. They are only indexed quantities. The indices are either contravariantly (upper indices) or covariantly (lower indices) placed. They can be symbolic or numeric. When a given index is found both in one upper and in one lower place, it is supposed to be summed over all space-coordinates it belongs to viz. it is a *dummy* index and *automatically recognized* as such. So they are supposed to obey the summation rules of tensor calculus. This why and only why they are called tensors. Moreover, aside from indices they may also depend implicitly or explicitly on any number of *variables*. Within this definition, tensors may also be spinors, they can be non-commutative or anticommutative, they may also be algebra generators and represent fields or quantum fields.

Implications of `TENSOR` declaration

The procedure `TENSOR` which takes an arbitrary number of identifiers as argument defines them as operator-like objects which admit an arbitrary number of indices. Each component has a formal character and may or may not belong to a specific (sub-)space. Numeric indices are also allowed. The way to distinguish upper and lower indices is the same as the one in the package `EXCALC` e.g. $-a$ is a lower index and a is an upper index. A special printing function has been created so as to mimic as much as possible the way of writing such objects on a sheet of paper. Let us illustrate the use of `TENSOR`:

```
   tensor te; ==> t

   te(3,a,-4,b,-c,7); ==>
                               3 a   b   7
                            te
                                  4   c



  te(3,a,{x,y},-4,b,-c,7); ==>

              3 a   b   7
           te            (x,y)
                 4   c



   te(3,a,-4,b,{u,v},-c,7); ==>

              3 a   b   7
           te            (u,v)
                 4   c


   te({x,y}); ==>  te(x,y)
```

Notice that the system distinguishes indices from variables on input solely on the basis that the user puts variables *inside a list*.

The dependence can also be declared implicit through the REDUCE command `DEPEND` which is generalized so as to allow to declare a tensor to depend on another tensor irrespective of its components. It means that only *one* declaration is enough to express the dependence with respect to *all its components*. A simple example:

```
tensor te,x;

depend te,x;

df(te(a,-b),x(c)); ==>

                a    c
       df(te     ,x )
                  b
```

Therefore, when *all* objects are tensors, the dependence declaration is valid for all indices.

One can also avoid the trouble to place the explicit variables inside a list if one declare them as variables through the command MAKE_VARIABLES. This property can also be removed[3] using REMOVE_VARIABLES:

```
make_variables x,y; ==> t

te(x,y); ==> te(x,y)


te(x,y,a); ==>

            a
       te    (x,y)


remove_variables x; ==> t


te(x,y,a); ==>

           x a
       te     (y)
```

If one does that one must be careful not to substitute a number to such declared

[3]One important feature of this package is its *reversibility* viz. it gives the user the means to erase its previous operations at any time. So, most functions described below do possess "removing" action companions.

variables because this number would be considered as an index and no longer as a variable. So it is only useful for *formal* variables.

A tensor can be easily eliminated using the function REM_TENSOR. It has the syntax

```
rem_tensor t1,t2,t3 ....;
```

Dummy indices recognition For all individual tensors met by the evaluator, the system will analyse the written indices and will detect those which must be considered dummy according to the usual rules of tensor calculus. Those indices will be given the dummy property and will no longer be allowed to play the role of *free* indices unless the user removes this dummy property. In that way, the system checks immediately the consistency of an input. Three functions are at the disposal of the user to control dummy indices. They are DUMMY_INDICES, REM_DUMMY_INDICES and REM_DUMMY_IDS. The following illustrates their use as well as the behaviour of the system:

```
dummy_indices(); ==> {} % In a fresh environment

te(a,b,-c,-a); ==>

         a b
       te
             c a

dummy_indices(); ==> {a}

te(a,b,-c,a); ==>

***** ((c)(a b a)) are inconsistent lists of indices

         % a cannot be found twice as an upper index

te(a,b,-b,-a); ==>

         a b
      te
             b a

dummy_indices(); ==> {b,a}

te(d,-d,d); ==>
```

```
     ***** ((d)(d d)) are inconsistent lists of indices

   dummy_indices(); ==> {d,b,a}

   rem_dummy_ids d; ==> t

  dummy_indices(); ==> {b,a}

   te(d,d); ==>

            d d
         te              %  This is allowed again.

  dummy_indices(); ==> {b,a}

   rem_dummy_indices(); ==> t

   dummy_indices(); ==> {}
```

Other verifications of coherence are made when space specifications are introduced both in the ON and OFF `onespace` environment. We shall discuss them later.

Substitutions, assignements and rewriting rules The user must be able to manipulate and give specific characteristics to the generic tensors he has introduced. Since tensors are essentially REDUCE operators, the usual commands of the system are available. However, some limitations are implied by the fact that indices and, especially numeric indices, must always be properly recognized before any substitution or manipulation is done. We have gathered below a set of examples which illustrate all the “delicate” points. First, the substitutions:

```
   sub(a=-c,te(a,b)); ==>

              b
          te
            c

   sub(a=-1,te(a,b)); ==>

              b
          te
            1

   sub(a=-0,te(a,b)); ==>
```

```
              0 b
            te    % sub has replaced -0 by 0. wrong!

   sub(a=-!0,te(a,b)); ==>

                b
            te    % right
              0
```

The substitution of an index by -0 is the *only one* case where there is a problem. The function SUB replaces -0 by 0 because it does not recognize 0 as an index of course. Such a recognition is context dependent and implies a modification of SUB for this *single* exceptional case. Therefore,we have opted, not do do so and to use the index 0 which is simply !0 instead of 0.

Second, the assignments. Here, we advise the user to rely on the operator==[4] instead of the operator `:=`. Again, the reason is to avoid the problem raised above in the case of substitutions. := does not evaluate its left hand side so that -0 is not recognized as an index and simplified to 0 while the == operator evaluates both its left and right hand sides and does recognize it. The disadvantage of == is that it demands that a second assignement on a given component be made only after having suppressed *explicitly* the first assignement. This is done by the function `REM_VALUE_TENS` which can be applied on any component. We stress, however, that if one is willing to use -!0 instead of -0 as the lower 0 index, the use of := is perfectly legitimate:

```
   te({x,y},a,-0)==x*y*te(a,-0); ==>

              a
            te   *x*y
                0

   te({x,y},a,-0); ==>

              a
            te   *x*y
                0

   te({x,y},a,0); ==>
```

[4]See the `ASSIST` documentation for its description.

```
              a 0
           te    (x,y)


      te({x,y},a,-0)==x*y*te(a,-0); ==>

                 a
          ***** te    *x*y invalid as setvalue kernel
                   0

      rem_value_tens te({x,y},a,-0);

      te({x,y},a,-0); ==>

              a
           te    (x,y)
                0

      te({x,y},a,-0)==(x+y)*te(a,-0); ==>

          a
        te   *(x + y)
            0
```

In the elementary application below, the use of a tensor avoids the introduction of two different operators and makes the calculation more readable.

```
      te(1)==sin th * cos phi; ==> cos(phi)*sin(th)

      te(-1)==sin th * cos phi; ==> cos(phi)*sin(th)

      te(2)==sin th * sin phi; ==> sin(phi)*sin(th)

      te(-2)==sin th * sin phi; ==> sin(phi)*sin(th)

      te(3)==cos th ; ==> cos(th)

      te(-3)==cos th ; ==> cos(th)

      for i:=1:3 sum te(i)*te(-i); ==>

          2         2           2            2        2
   cos(phi) *sin(th)  + cos(th)  + sin(phi) *sin(th)
```

```
      rem_value_tens te;

      te(2); ==>

                2
             te
```

There is no difference in the manipulation of numeric indices and numeric *tensor* indices. The function `REM_VALUE_TENS` when applied to a tensor prefix suppresses the value of *all its components*. Finally, there is no “interference” with i as a dummy index and i as a numeric index in a loop.

Third, rewriting rules. They are either global or local and can be used as in REDUCE. Again, here, the -0 index problem exists each time a substitution by the index -0 must be made in a template.

```
  % LET:

      let te({x,y},-0)=x*y;

      te({x,y},-0); ==> x*y

      te({x,y},+0); ==>

                 0
              te  (x,y)


      te({x,u},-0); ==>

              te  (x,u)
                 0

  % FOR ALL .. LET:

      for all x,a let te({x},a,-b)=x*te(a,-b);

      te({u},1,-b); ==>

                1
              te   *u
                 b

      te({u},c,-b); ==>
```

```
             c
           te   *u
               b

   te({u},b,-b); ==>

             b
           te   *u
               b

   te({u},a,-a); ==>

              a
           te    (u)
                a

   for all x,a clear te({x},a,-b);

   te({u},c,-b); ==>

              c
           te    (u)
                b

   for all a,b let te({x},a,-b)=x*te(a,-b);

   te({x},c,-b); ==>

             c
           te   *x
               b

   te({x},a,-a); ==>

             a
           te   *x
               a

% The index -0 problem:

   te({x},a,-0); ==> % -0 becomes +0 in the template

              a
```

```
         te    (x)  %  the rule does not apply.
              0

   te({x},0,-!0); ==>

            0
         te   *x      % here it applies.
             0

% WHERE:

   rul:={te(~a) => sin a}; ==>

                    a
         rul := {te  => sin(a)}



   te(1) where rul; ==> sin(1)

   te(1); ==>

             1
           te


% with variables:

   rul1:={te(~a,{~x,~y}) => x*y*sin(a)}; ==>


                    ~a
         rul1 := {te   (~x,~y) => x*y*sin(a)}


   te(a,{x,y}) where rul1; ==> sin(a)*x*y



   te({x,y},a) where rul1; ==> sin(a)*x*y


   rul2:={te(-~a,{~x,~y}) => x*y*sin(-a)};
```

```
          rul2 := {te   (~x,~y) => x*y*sin(-a)}
                     ~a

    te(-a,{x,y}) where rul2; ==> -sin(a)*x*y


    te({x,y},-a) where rul2; ==> -sin(a)*x*y
```

Notice that the position of the list of variables inside the rule may be chosen at will. It is an irrelevant feature of the template. This may be confusing, so, we advise to write the rules not as above but placing the list of variables *in front of all indices* since it is in that canonical form which it is written by the simplification function of individual tensors.

Behaviour under space specifications

The characteristics and the behaviour of generic tensors described up to now are independent of all space specifications. They are complete as long as we confine to the default space which is active when starting CANTENS. However, as soon as some space specification is introduced, it has some consequences one the generic tensor properties. This is true both when ONESPACE is switched ON or OFF. Here we shall describe how to deal with these features.

When onespace is ON, if the space dimension is set to an integer, numeric indices of any generic tensors are forced to be less or equal that integer if the signature is 0 or less than that integer if the signature is equal to 1. The following illustrates what happens.

```
    on onespace;

    wholespace_dim 4; ==> 4

    signature 0; ==> 0

    te(3,a,-b,7); ==> ***** numeric indices out of range

    te(3,a,-b,3); ==>


              3 a   3
            te
                 b
```

```
      te(4,a,-b,4); ==>


                4 a   4
              te
                    b

      sub(a=5,te(3,a,-b,3));

                   ==> ***** numeric indices out of range

      signature 1; ==> 1

  % Now indices range from 0 to 3:

      te(4,a,-b,4);

                    ==> ***** numeric indices out of range

      te(0,a,-b,3); ==>


                 0 a   3
               te
                     b
```

When `onespace` is OFF, many more possibilities to control the input or to give specific properties to tensors are open. For instance, it is possible to declare that a tensor belongs to one of them. It is also possible to declare that some indices belongs to one of them. It is even possible to do that for *numeric* indices thanks to the declaration indexrange included optionally in the space definition generated by DEFINE_SPACES. First, when `onespace` is OFF, the run equivalent to the previous one is like the following:

```
      off onespace;

      define_spaces wholespace={6,signature=1); ==> t

      show_spaces(); ==> {{wholespace,6,signature=1}}

      make_tensor_belong_space(te,wholespace);
```

```
                                   ==> wholespace

te(4,a,-b,6); ==>

               ***** numeric indices out of range

te(4,a,-b,5); ==>


         4 a   5
       te
             b

rem_spaces wholespace;

define_spaces wholespace={4,euclidean}; ==> t

te(a,5,-b); ==> ***** numeric indices out of range

te(a,4,-b); ==>


         a 4
       te
             b

define_spaces eucl={1,signature=0}; ==> t

show_spaces(); ==>

  {{wholespace,5,signature=1},

         {eucl,1,signature=0}}

make_tensor_belong_space(te,eucl); ==> eucl

te(1); ==>

         1
       te

te(2); ==> ***** numeric indices out of range
te(0); ==>

          0
        te
```

In the run, the new function MAKE_TENSOR_BELONG_SPACE has been used. One may be surprised that te(0) is allowed in the end of the previous run and, indeed, it is incorrect that the system allows *two* different components to te. This is due to an incomplete definition of the space. When one deals with spaces of integer dimensions, if one wants to control numeric indices correctly *when* onespace is switched off *one must also give the indexrange*. So the previous run must be corrected to

```
define_spaces eucl=

                {1,signature=0,indexrange=1 .. 1}; ==> t

make_tensor_belong_space(te,eucl); ==> eucl

te(0); ==>

  ***** numeric indices do not belong to (sub)-space

te(1); ==>

          1
        te

te(2); ==>

  ***** numeric indices do not belong to (sub)-space
```

Notice that the error message has also changed accordingly. So, now one can even constrain the 0 component to belong to an euclidian space.

Let us go back to symbolic indices. By default, any symbolic index belongs to the global space or to all defined partial spaces. In many cases, this is, of course, not consistent. So, the possibility exists to declare that one or several indices belong to a specific (sub-)space. To this end, one is to use the function MK_IDS_BELONG_SPACE. Its syntax is

```
mk_ids_belong_space(<list of indices>,
                                  <(sub-)space identifier>)
```

The function `MK_IDS_BELONG_ANYSPACE` whose syntax is the same do the reverse operation.

Combined with the declaration `MAKE_TENSOR_BELONG_SPACE`, it allows to express all problems which involve tensors belonging to different spaces and do the dummy summations correctly. One can also define a tensor which has a "bloc-diagonal" structure. All these features are illustrated in the next sections which describe specific tensors and the properties of the extended function `CANONICAL`.

16.11.4 Specific tensors

The means provided in the two previous subsection to handle generic tensors already allow to construct any specific tensor we may need. That the package contains a certain number of them is already justified on the level of conviviality. However, a more important justification is that some basic tensors are so universaly and frequently used that a careful programming of these improves considerably the robustness and the efficiency of most calculations. The choice of the set of specific tensors is not clearcut. We have tried to keep their number to a minimum but, experience, may lead us extend it without dificulty. So, up to now, the list of specific tensors is:

- `delta` tensor,
- `eta` Minkowski tensor,
- `epsilon` tensor,
- `del` generalised delta tensor,
- `metric` generic tensor metric.

It is important to realize that the typewriter font names in the list are *keywords* for the corresponding tensors and do not necessarily correspond to their *actual names*. Indeed, the choice of the names of particular tensors is left to the user. When startting `CANTENS` specific tensors are NOT available. They must be activated by the user using the function `MAKE_PARTIC_TENS` whose syntax is:

```
make_partic_tens(<tensor name> , <keyword>);
```

The name chosen may be the same as the keyword. As we shall see, it is never needed to define more than one `delta` tensor but it is often needed to define several `epsilon` tensors. Hereunder, we describe each of the above tensors especially their behaviour in a multi-space environment.

DELTA tensor

It is the simplest example of a bloc-diagonal tensor we mentioned in the previous section. It can also work in a space which is a direct product of two spaces. Therefore, one never needs to introduce more than one such tensor. If one is working in a graphic environment, it is advantageous to choose the keyword as its name. Here we choose `DELT`. We illustrate how it works when the switch `onespace` is successively switched ON and OFF.

```
      on onespace;

      make_partic_tens(delt,delta); ==> t

      delt(a,b); ==>

          ***** bad choice of indices for DELTA tensor

  % order of upper and lower indices irrelevant:

      delt(a,-b); ==>

                 a
             delt
                 b

      delt(-b,a); ==>

                 a
             delt
                 b

      delt(-a,b); ==>

                 b
             delt
                 a


      wholespace_dim ?; ==> dim

      delt(1,-5); ==> 0

  % dummy summation done:
    delt(-a,a); ==> dim


    wholespace_dim 4; ==> 4

    delt(1,-5); ==> ***** numeric indices out of range


    wholespace_dim 3; ==> 3

    delt(-a,a); ==> 3
```

There is a peculiarity of this tensor, viz. it can serve to represent the Dirac *delta function* when it has no indices and an explicit variable dependency as hereunder

```
    delt({x-y}) ==> delt(x-y)
```

Next we work in the context of several spaces:

```
    off onespace;

    define_spaces wholespace={5,signature=1}; ==> t

  % we need to assign delta to wholespace when it exists:

    make_tensor_belong_space(delt,wholespace);

    delt(a,-a); ==> 5

    delt(0,-0); ==>1

    rem_spaces wholespace; ==> t

    define_spaces wholespace={5,signature=0}; ==> t

    delt(a,-a); ==> 5

    delt(0,-a); ==>

        ***** bad value of indices for DELTA tensor
```

The checking of consistency of chosen indices is made in the same way as for generic tensor. In fact, all the previous functions which act on generic tensors may also affect, in the same way, a specific tensor. For instance, it was compulsory to explicitly tell that we want DELT to belong to the wholespace overwise, DELT would remain defined on the *default space*. In the next sample run, we display the bloc-diagonal property of the delta tensor.

```
      onespace ?; ==> no

      rem_spaces wholespace; ==> t

      define_spaces wholespace={10,signature=1}$

      define_spaces d1={5,euclidian}$

      define_spaces d2={2,euclidian}$


      mk_ids_belong_space({a},d1); ==> t

      mk_ids_belong_space({b},d2); ==> t

  % c belongs to wholespace so:

      delt(c,-b); ==>

                  c
             delt
                  b

      delt(c,-c); ==> 10


      delt(b,-b); ==> 2

      delt(a,-a); ==> 5

  % this is especially important:

      delt(a,-b); ==> 0
```

The bloc-diagonal property of `delt` is made active under two conditions. The first is that the system knows to which space it belongs, the second is that indices must be declared to belong to a specific space. To enforce the same property on a generic tensor, we have to make the MAKE_BLOC_DIAGONAL declaration:

```
      make_bloc_diagonal t1,t2, ...;
```

and to make it active, one proceeds as in the above run. Starting from a fresh environment, the following sample run is illustrative:

```
      off onespace;

      define_spaces wholespace={6,signature=1}$

      define_spaces mink={4,signature=1,indexrange=0 .. 3}$

      define_spaces eucl={3,euclidian,indexrange=4 .. 6}$

      tensor te;

      make_tensor_belong_space(te,eucl); ==> eucl


  % the key declaration:

      make_bloc_diagonal te; ==> t


  % bloc-diagonal property activation:

      mk_ids_belong_space({a,b,c},eucl); ==> t

      mk_ids_belong_space({m1,m2},mink); ==> t


      te(a,b,m1); ==> 0

      te(a,b,m2); ==> 0

   % bloc-diagonal property suppression:

      mk_ids_belong_anyspace a,b,c,m1,m2; ==> t


      te(a,b,m2); ==>

               a b m2
             te
```

ETA Minkowski tensor

The use of MAKE_PARTIC_TENS with the keyword eta allows to create a Minkowski diagonal metric tensor in a one or multi-space context either with the convention of high energy physicists or in the convention of astrophysicists. Any eta-like tensor is assumed to work within a space of signature 1. Therefore, if the space whose metric, it is supposed to describe has a signature 0, an error message follows if one is working in an ON onespace context and a warning when in an OFF onespace context. Illustration:

```
      on onespace;

      make_partic_tens(et,eta); ==> t

      signature 0; ==> 0;

      et(-b,-a); ==>

         *****  signature must be equal to 1 for ETA tensor


      off onespace;

      et(a,b); ==>

          ***  ETA tensor not properly assigned to a space

  % it is then evaluated to zero:

          0

      on onespace;

      signature 1; ==> 1

      et(-b,-a); ==>

            et
              a b
```

Since et(a,-a) is evaluated to the corresponding delta tensor, one cannot define properly an eta tensor without a simultaneous introduction of a delta tensor. Otherwise one gets the following message:

```
et(a,-a); ==> *****  no name found for (delta)
```

So we need to issue, for instance,

```
make_partic_tens(delta,delta); ==> t
```

The value of its diagonal elements depends on the chosen global sign. The next run illustrates this:

```
global_sign ?; ==> 1

et(0,0); ==> 1

et(3,3); ==>  - 1

global_sign(-1); ==> -1

et(0,0); ==>  - 1

et(3,3); ==> 1
```

The tensor is of course symmetric . Its indices are checked in the same way as for a generic tensor. In a multi_space context, the `eta` tensor must belong to a well defined space of signature 1:

```
off onespace;

define_spaces wholespace={4,signature=1}$

make_tensor_belong_space(et,wholespace)$

et(a,-a); ==> 4
```

If the space to which `et` belongs to is a subspace, one must also take care to give a space-identity to dummy indices which may appear inside it. In the following run, the index `a` belongs to `wholespace` if it is not told to the system that it is a dummy index of the space `mink`:

```
make_tensor_belong_anyspace et; ==> t

rem_spaces wholespace; ==> t

define_spaces wholespace={8,signature=1}; ==> t
```

```
      define_spaces mink={5,signature=1}; ==> t

      make_tensor_belong_space(et,mink); ==> mink

  % a sits in wholespace:

      et(a,-a); ==> 8

      mk_ids_belong_space({a},mink); ==> t

  % a sits in mink:

      et(a,-a); ==>  5
```

EPSILON tensors

It is an antisymmetric tensor which is the invariant tensor for the unitary group transformations in n-dimensional complex space which are continuously connected to the identity transformation. The number of their indices are always stricty equal to the number of space dimensions. So, to each specific space is associated a specific `epsilon` tensor. Its properties are also dependent on the signature of the space. We describe how to define and manipulate it in the context of a unique space and, next, in a multi-space context.

`ONESPACE` **is ON** The use of `MAKE_PARTIC_TENS` places it, by default, in an euclidian space if the signature is 0 and in a Minkowski-type space if the signature is 1. For higher signatures it is not constructed. For a space of symbolic dimension, the number of its indices is not constrained. When it appears inside an expression, its indices are *all* currently upper or lower indices. However, the system allows for mixed positions of the indices. In that case, the output of the system is changed compared to the input only to place all contravariant indices to the left of the covariant ones.

```
      make_partic_tens(eps,epsilon); ==> t

      eps(a,d,b,-g,e,-f); ==>

                   a d b e
             - eps
                           g f

      eps(a,d,b,-f,e,-f); ==> 0
```

```
% indices have all the same variance:

     eps(-b,-a); ==>

             - eps
                  a b


     signature ?; ==> 0

     eps(1,2,3,4); ==> 1

     eps(-1,-2,-3,-4); ==> 1

     wholespace_dim 3; ==> 3

     eps(-1,-2,-3); ==> 1

     eps(-1,-2,-3,-4); ==>

            ***** numeric indices out of range

     eps(-1,-2,-3,-3); ==>

            ***** bad number of indices for (eps) tensor

     eps(a,b); ==>

            ***** bad number of indices for (eps) tensor

     eps(a,b,c); ==>

               a b c
            eps


     eps(a,b,b); ==> 0
```

When the signature is equal to 1, it is known that there exists two *conventions* which are linked to the chosen value 1 or -1 of the $(0, 1, \ldots, n)$ component. So, the sytem does evaluate all components in terms of the $(0, 1, \ldots, n)$ upper index component. It is left to the user to assign it to 1 or -1.

```
    signature 1; ==> 1

    eps(0,1,2); ==>

             0 1 2
          eps

    eps(-0,-1,-2); ==>

             0 1 2
          eps

    wholespace_dim 4; ==> 4

    eps(0,1,2,3); ==>

             0 1 2 3
          eps

    eps(-0,-1,-2,-3); ==>

                0 1 2 3
           - eps

% change of the global_sign convention:

    global_sign(-1);

    wholespace_dim 3; ==> 3

% compare with second input:

    eps(-0,-1,-2); ==>

                0 1 2
           - eps
```

`ONESPACE` **is OFF** As already said, several epsilon tensors may be defined. They *must* be assigned to a well defined (sub-)space otherwise the simplifying function `CANONICAL` will not properly work. The set of epsilon tensors defined associated to their space-name may be retrieved using the function `SHOW_EPSILONS`. An important word of caution here. The output of this function does NOT show the epsilon tensor one may have defined in the ON `onespace` context. This is so because the default space has *NO* name. Starting from a fresh environment, the following run illustrates this point:

```
show_epsilons(); ==> {}

onespace ?; ==> yes

make_partic_tens(eps,epsilon); ==> t

show_epsilons(); ==> {}
```

To make the `epsilon` tensor defined in the single space environment visible in the multi-space environment, one needs to associate it to a space. For example:

```
off onespace;

define_spaces wholespace={7,signature=1}; ==> t

show_epsilons(); ==> {}   % still invisible

make_tensor_belong_space(eps,wholespace); ==>

                                       wholespace

show_epsilons(); ==> {{eps,wholespace}}
```

Next, let us define an *additional* `epsilon`-type tensor:

```
define_spaces eucl={3,euclidian}; ==> t

make_partic_tens(ep,epsilon); ==>

          *** Warning: ep MUST belong to a space
            t

make_tensor_belong_space(ep,eucl); ==> eucl


show_epsilons(); ==> {{ep,eucl},{eps,wholespace}}

% We show that it is indeed working inside eucl:

ep(-1,-2,-3); ==> 1
ep(1,2,3); ==> 1

ep(a,b,c,d); ==>

      ***** bad number of indices for  (ep)  tensor

ep(1,2,4); ==>

      ***** numeric indices out of range
```

As previously, the discrimation between symbolic indices may be introduced by assigning them to one or another space :

```
rem_spaces wholespace;

define_spaces wholespace={dim,signature=1}; ==> t

mk_ids_belong_space({e1,e2,e3},eucl); ==> t

mk_ids_belong_space({a,b,c},wholespace); ==> t

ep(e1,e2,e3); ==>

         e1 e2 e3
       ep           % accepted

ep(e1,e2,z); ==>

         e1 e2 z
       ep            % accepted because z
                     % not attached to a space.

ep(e1,e2,a);==>

    ***** some indices are not in the space of ep


eps(a,b,c); ==>

          a b c
       eps           % accepted because *symbolic*
                     % space dimension.
```

epsilon-like tensors can also be defined on disjoint spaces. The subsequent sample run starts from the environment of the previous one. It suppresses the space wholespace as well as the space-assignment of the indices a,b,c. It defines the new space mink. Next, the previously defined eps tensor is attached to this space. ep remains unchanged and e1,e2,e3 still belong to the space eucl.

```
rem_spaces wholespace; ==> t

make_tensor_belong_anyspace eps; ==> t

show_epsilons(); ==> {{ep,eucl}}

show_spaces(); ==> {{eucl,3,signature=0}}

mk_ids_belong_anyspace a,b,c; ==> t

define_spaces mink={4,signature=1}; ==> t

show_spaces(); ==>

       {{eucl,3,signature=0},

        {mink,4,signature=1}}

make_tensor_belong_space(eps,mink); ==> mink

show_epsilons(); ==> {{eps,mink},{ep,eucl}}

eps(a,b,c,d); ==>

       a b c d
    eps

eps(e1,b,c,d); ==>

   ***** some indices are not in the space of eps

ep(e1,b,c,d); ==>

   ***** bad number of indices for  (ep)  tensor

ep(e1,b,c); ==>

         b c e1
         ep


   ep(e1,e2,e3); ==>

           e1 e2 e3
         ep
```

DEL generalized delta tensor

The generalized delta function comes from the contraction of two epsilons. It is totally antisymmetric. Suppose its name has been chosen to be gd, that the space to which it is attached has dimension n while the name of the chosen delta tensor is δ, then one can define it as follows:

$$gd^{a_1,a_2,\ldots,a_n}_{b_1,b_2,\ldots,b_n} = \begin{vmatrix} \delta^{a_1}_{b_1} & \delta^{a_1}_{b_2} & \ldots & \delta^{a_1}_{b_n} \\ \delta^{a_2}_{b_1} & \delta^{a_2}_{b_2} & \ldots & \delta^{a_2}_{b_n} \\ \vdots & \vdots & \ddots & \vdots \\ \delta^{a_n}_{b_1} & \delta^{a_n}_{b_1} & \ldots & \delta^{a_n}_{b_1} \end{vmatrix}$$

It is, in general uneconomical to explicitly write that determinant except for particular *numeric* values of the indices or when almost all upper and lower indices are recognized as dummy indices. In the sample run below, gd is defined as the generalized delta function in the default space. The main automatic evaluations are illustrated. The indices which are summed over are always simplified:

```
   onespace ? ==> yes

   make_partic_tens(delta,delta); ==> t

   make_partic_tens(gd,del); ==> t

% immediate simplifications:

   gd(1,2,-3,-4); ==> 0

   gd(1,2,-1,-2); ==> 1

   gd(1,2,-2,-1); ==> -1   % antisymmetric

   gd(a,b,-a,-b);

         ==> dim*(dim - 1) % summed over dummy indices
```

```
      gd(a,b,c,-a,-d,-e); ==>

                  b c
                gd    *(dim - 2)
                  d e

      gd(a,b,c,-a,-d,-c); ==>

                     b      2
                delta  *(dim  - 3*dim + 2)
                     d

  % no simplification:

      gd(a,b,c,-d,-e,-f); ==>

                  a b c
                gd
                  d e f
```

One can force evaluation in terms of the determinant in all cases. To this end, the switch EXDELT is provided. It is initially OFF. Switching it ON will most often give inconveniently large outputs:

```
      on exdelt;

      gd(a,b,c,-d,-e,-f); ==>

           a       b       c          a       b       c
      delta  *delta  *delta   - delta  *delta  *delta
           d       e       f          d       f       e

              a       b       c          a       b       c
       - delta  *delta  *delta   + delta  *delta  *delta
              e       d       f          e       f       d

              a       b       c          a       b       c
       + delta  *delta  *delta   - delta  *delta  *delta
              f       d       e          f       e       d
```

In a multi-space environment, it is never necessary to define several such tensor. The reason is that CANONICAL uses it always from the contraction of a pair of epsilon-like tensors. Therefore the control of indices is already done, the space-dimension in which del is working is also well defined.

`METRIC` tensors

Very often, one has to define a specific metric. The `metric`-type of tensors include all generic properties. The first one is their symmetry, the second one is their equality to the `delta` tensor when they get mixed indices, the third one is their optional bloc-diagonality. So, a metric (generic) tensor is generated by the declaration

```
make_partic_tens(<tensor-name>,metric);
```

By default, when one is working in a multi-space environment, it is defined in `wholespace` One uses the usual means of REDUCE to give it specific values. In particular, the metric 'delta' tensor of the euclidian space can be defined that way. Implicit or explicit dependences on variables are allowed. Here is an illustration in the single space environment:

```
make_partic_tens(g,metric); ==> t

make_partic_tens(delt,delta); ==> t

onespace ?; ==> yes

g(a,b); ==>

             a b
            g

g(b,a); ==>

             a b
            g

g(a,b,c); ==>

   ***** bad choice of indices for a METRIC tensor


g(a,b,{x,y}); ==>

              a b
            g    (x,y)

g(a,-b,{x,z}); ==>
```

```
              a
          delt
              b


let g({x,y},1,1)=1/2(x+y);


g({x,y},1,1); ==>

           x + y
          -------
             2


rem_value_tens g({x,y},1,1);


g({x,y},1,1); ==>

           1 1
          g    (x,y)
```

16.11.5 The simplification function CANONICAL

Tensor expressions

Up to now, we have described the behaviour of individual tensors and how they simplify themselves whenever possible. However, this is far from being sufficient. In general, one is to deal with objects which involve several tensors together with various dummy summations between them. We define a tensor expression as an arbitrary multivariate polynomial. The indeterminates of such a polynomial may be either an indexed object, an operator, a variable or a rational number. A tensor-type indeterminate cannot appear to a degree larger than one except if it is a trace. The following is a tensor expression:

```
aa:= delt({x - y})*delt(a, - g)*delt(d, - g)*delt(g, -r)

      *eps( - d, - e, - f)*eps(a,b,c)*op(x,y) + 1; ==>


                          a      d      g
```

```
          aa := delt(x - y)*delt   *delt   *delt   *eps
                                g       g       r     d e f


                         a b c
                     *eps     *op(x,y) + 1
```

In the above expression, `delt` and `eps` are, respectively, the `delta` and the `epsilon` tensors, `op` is an operator. and `delt(x-y)` is the Dirac delta function. Notice that the above expression is not cohérent since the first term has a variance while the second term is a scalar. Moreover, the dummy index `g` appears *three* times in the first term. In fact, on input, each factor is simplified and each factor is checked for coherence not more. Therefore, if a dummy summation appears inside one factor, it will be done whenever possible. Hereunder `delt(a,-a)` is summed over:

```
   sub(g=a,aa); ==>

                          a      d              a b c
          delt(x - y)*delt   *delt   *eps     *eps
                          r      a     d e f

                      *op(x,y)*dim + 1
```

The use of `CANONICAL`

`CANONICAL` is an offspring of the function with the same name of the package `DUMMY`. It applies to tensor expressions as defined above. When it acts, this functions has several features which are worth to realise:

1. It tracks the free indices in each term and checks their identity. It identifies and verify the coherence of the various dummy index summations.
2. Dummy indices summations are done on tensor products whenever possible since it recognises the particular tensors defined above or defined by the user.
3. It seeks a canonical form for the various simplified terms, makes the comparison between them. In that way it maximises simplifications and generates a canonical form for the output polynomial.

Its capabilities have been extended in four directions:

- It is able to work within *several* spaces.

- It manages correctly expressions where formal tensor *derivatives* are present[5].
- It takes into account all symmetries even if partial.
- As its parent function, it can deal with non-commutative and anticommutative indexed objects. So, Indexed objects may be spinors or quantum fields.

We describe most of these features in the rest of this documentation.

Check of tensor indices

Dummy indices for individual tensors are kept in the memory of the system. If they are badly distributed over several tensors, it is CANONICAL which gives an error message:

```
tensor te,tf; ==> t

bb:=te(a,b,b)*te(-b); ==>

                    a b b
       bb := te *te
                b


canonical bb; ==>

***** ((b)(a b b)) are inconsistent lists of indices


aa:=te(b,-c)*tf(b,-c); ==>

               b     b
       aa := te   *tf     % b and c are free.
                 c     c

canonical aa; ==>

         b     b
       te   *tf
           c     c

bb:=te(a,c,b)*te(-b)*tf(a)$
```

[5]In DUMMY it does not take them into account

```
canonical bb; ==>


          a c     b   a
        te     *te *tf
               b

delt(a,-a); ==> dim  % a is now a dummy index


canonical bb; ==>

        ***** wrong use of indices (a)
```

The message of canonical is clear, the first sublist contains the list of all lower indices, and the second one the list of all upper indices. The index `b` is repeated *three* times. In the second example, `b` and `c` are considered as free indices, so they may be repeated. The last example shows the interference between the check on individual tensors and the one of canonical. The use of `a` as dummy index inside `delt` does no longer allow `a` to be used as a free index in expression `bb`. To be usable, one must explicitly remove it as dummy index using `REM_DUMMY_INDICES`. Dans le quatrième cas, il n'y a pas de problème puisque `b` et `c` sont tous les deux des indices *libres*. `CANONICAL` checks that in a tensor polynomial all do possess the *same* variance:

```
aa:=te(a,c)+x^2; ==>

              a c    2
        aa := te    + x

canonical aa; ==>

        ***** scalar added with tensor(s)

aa:=te(a,b)+tf(a,c); ==>

              a b     a c
        aa := te    + tf

canonical aa; ==>

       ***** mismatch in free indices :  ((a c) (a b))
```

In the message the first two lists of incompatible indices are explicitly indicated. So, it is not an exhaustive message and a more complete correction may be needed. Of course, no message of that kind appears if the indices are inside ordinary operators[6]

```
dummy_names b; ==> t

cc:=op(b)*op(a,b,b); ==> cc := op(a,b,b)*op(b)

canonical cc; ==> op(a,b,b)*op(b)

clear_dummy_names; ==> t
```

Position and renaming of dummy indices

For a specific tensor, contravariant dummy indices are place in front of covariant ones. This already leads to some useful simplifications. For instance:

```
pp:=te(a,-a)+te(-a,a)+1; ==>

                a     a
    pp := te     + te     + 1
              a         a

canonical pp; ==>

          a
    2*te     + 1
            a

pp:=te(a,-a)+te(-b,b); ==>

                b     a
    pp := te     + te
              b         a


canonical pp; ==>

          a
    2*te
            a
```

[6]This is the case inside the DUMMY package.

```
     pp:=te(r,a,c,d,-a,f)+te(r,-b,c,d,b,f); ==>

                   r   c d b f     r a c d   f
           pp := te            + te
                     b                     a

  canonical pp; ==>


            r a c d   f
         2*te
                 a
```

In the second and third example, there is also a renaming of the dummy variable b whih becomes a. There is a loophole at this point. For some expressions one will never reach a stable expression. This is the case for the following very simple monom:

```
tensor nt; ==> t

a1:=nt(-a,d)*nt(-c,a); ==>

          d     a
       nt   *nt
         a     c

canonical a1; ==>


              a d
       nt   *nt
         c a

a12:=a1-canonical a1; ==>

                d     a            a d
       a12 := nt   *nt     - nt   *nt
                 a     c       c a


canonical a12; ==>
```

```
              d     a           a d
        - nt   *nt    + nt   *nt   % changes sign.
             a     c       c a
```

In the above example, no canonical form can be reached. When applied twice on the tensor monom `a1` it gives back `a1`!

No change of dummy index position is allowed if a tensor belongs to an `AFFINE` space. With the tensor polynomial `pp` introduced above one has:

```
off onespace;

define_spaces aff={dd,affine}; ==> t

make_tensor_belong_space(te,aff); ==> aff

mk_ids_belong_space({a,b},aff); ==> t

canonical pp; ==>

          r   c d a f     r a c d   f
        te             + te
             a                    a
```

The renaming of `b` has been made however.

Contractions and summations with particular tensors

This is a central part of the extension of `CANONICAL`. The required contractions and summations can be done in a multi-space environment as well in a single space environment.

The case of `DELTA`

Dummy indices are recognized contracted and summed over whenever possible:

```
aa:=delt(a,-b)*delt(b,-c)*delt(c,-a) + 1; ==>

                  a      b      c
        aa := delt   *delt  *delt   + 1
                  b      c      a
```

```
canonical aa; ==> dim + 1

aa:=delt(a,-b)*delt(b,-c)*delt(c,-d)*te(d,e)$

canonical aa; ==>

          a e
        te
```

CANONICAL will not attempt to make contraction with dummy indices included inside ordinary operators:

```
operator op;

aa:=delt(a,-b)*op(b,b)$


canonical aa; ==>

           a
       delt  *op(b,b)
           b

dummy_names b; ==> t

canonical aa; ==>

          a
     delta  *op(b,b)
          b
```

The case of ETA

First, we introduce ETA:

```
make_partic_tens(eta,eta); ==> t

signature 1; ==> 1 % necessary

aa:=delta(a,-b)*eta(b,c); ==>

                  a     b c
        aa := delt  *eta
```

```
                  b

canonical aa; ==>

           a c
        eta


canonical(eta(a,b)*eta(-b,c)); ==>

           a c
        eta


canonical(eta(a,b)*eta(-b,-c)); ==>

            a
        delt
            c

canonical(eta(a,b)*eta(-b,-a)); ==> dim

canonical (eta(-a,-b)*te(d,-e,f,b)); ==>

          d   f
        te
            e   a


aa:=eta(a,b)*eta(-b,-c)*te(-a,c)+1; ==>

                        a b     c
        aa := eta   *eta   *te    + 1
                 b c          a

canonical aa; ==>

          a
        te    + 1
            a

aa:=eta(a,b)*eta(-b,-c)*delta(-a,c)+

    1+eta(a,b)*eta(-b,-c)*te(-a,c)$
```

```
      canonical aa; ==>

                 a
              te    + dim + 1
                  a
```

Let us add a generic metric tensor:

```
      aa:=g(a,b)*g(-b,-d); ==>

                         a b
              aa := g   *g
                     b d

      canonical aa; ==>

                  a
              delt
                  d

      aa:=g(a,b)*g(c,d)*eta(-c,-e)*eta(e,f)*te(-f,g); ==>

                              e f  a b  c d     g
              aa := eta   *eta   *g   *g   *te
                        c e                   f

      canonical aa; ==>

               a b   d g
              g   *te
```

The case of EPSILON

The epsilon tensor plays an important role in many contexts. CANONICAL realises the contraction of two epsilons if and only if they belong to the same space. The proper use of CANONICAL on expressions which contains it requires a preliminary definition of the tensor DEL. When the signature is 0; the contraction of two epsilons gives a DEL-like tensor. When the signature is equal to 1, it is equal to *minus* a DEL-like tensor. Here we choose 1 for the signature and we work in a single space. We define the DEL tensor:

```
on onespace;

wholespace_dim dim; ==> dim

make_partic_tens(gd,del); ==> t

signature 1; ==> 1
```

We define the EPSILON tensor and show how CANONICAL contracts expression containing *two*[7] of them:

```
aa:=eps(a,b)*eps(-c,-d); ==>

                           a b
       aa := eps    *eps
                c d

canonical aa; ==>

              a b
        - gd
              c d


aa:=eps(a,b)*eps(-a,-b); ==>

                           a b
       aa := eps    *eps
                a b

canonical aa; ==> dim*( - dim + 1)

on exdelt;

gd(-a,-b,a,b); ==> dim*(dim - 1)

aa:=eps(a,b,c)*eps(-b,-d,-e)$

canonical aa; ==>

                 a       c                      a       c
```

[7]No contractions are done on expressions containing three or more epsilons which sit in the *same* space. We are not sure whether it is useful to be more general than we are presently.

```
        delt  *delt  *dim - 2*delt  *delt  -
            d      e              d      e

              a      c               a       c
        - delt  *delt  *dim + 2*delt  * delt
              e      d               e       d
```

Several expressions which contain the epsilon tensor together with other special tensors are given below as examples to treat with CANONICAL:

```
aa:=eps( - b, - c)*eta(a,b)*eta(a,c); ==>

                 a b    a c
      eps   *eta   *eta
         b c

canonical aa; ==> 0

aa:=eps(a,b,c)*te(-a)*te(-b); ==> % te is generic.

                a b c
      aa := eps     *te *te
                       a   b

canonical aa; ==> 0

tensor tf,tg;

aa:=eps(a,b,c)*te(-a)*tf(-b)*tg(-c)

    + eps(d,e,f)*te(-d)*tf(-e)*tg(-f); ==>

 canonical aa; ==>

            a b c
      2*eps     *te *tf *tg
                   a   b   c

aa:=eps(a,b,c)*te(-a)*tf(-c)*tg(-b)

    + eps(d,e,f)*te(-d)*tf(-e)*tg(-f)$

canonical aa; ==> 0
```

Since CANONICAL is able to work inside several spaces, we can introduce also several epsilons and make the relevant simplifications on each (sub)-spaces. This is the goal of the next illustration.

```
off onespace;

define_spaces wholespace=

                  {dim,signature=1}; ==> t

define_spaces subspace=

                   {3,signature=0}; ==> t

show_spaces(); ==>

       {{wholespace,dim,signature=1},

        {subspace,3,signature=0}}

make_partic_tens(eps,epsilon); ==> t

make_partic_tens(kap,epsilon); ==> t

make_tensor_belong_space(eps,wholespace);

                        ==> wholespace

make_tensor_belong_space(kap,subspace);

                        ==> subspace

show_epsilons(); ==>

             {{eps,wholespace},{kap,subspace}}

off exdelt;

aa:=kap(a,b,c)*kap(-d,-e,-f)*eps(i,j)*eps(-k,-l)$

canonical aa; ==>

            a b c    i j
        - gd      *gd
```

```
                    d e f    k l
```

If there are no index summation, as in the expression above, one can develop both terms into the delta tensor with `EXDELT` switched ON. In fact, the previous calculation is correct *only if there are no dummy index* inside the two `gd`'s. If some of the indices are dummy, then we must take care of the respective spaces in which the two `gd` tensors are considered. Since, the tensor themselves do not belong to a given space, the space identification can only be made through the indices. This is enough since the `DELTA`-like tensor is bloc-diagonal. With `aa` the result of the above illustration, one gets, for example,:

```
mk_ids_belong_space({a,b,c,d,e,f},wholespace)$

mk_ids_belong_space({i,j,k,l},subspace)$

sub(d=a,e=b,k=i,aa); ==>

                 c      j         2
          2*delt  *delt  *( - dim  + 3*dim - 2)
                 f      l

sub(k=i,l=j,aa); ==>
                                  a b c
                            - 6*gd
                                  d e f
```

`CANONICAL` and symmetries

Most of the time, indexed objects have some symmetry property. When this property is either full symmetry or antisymmetry, there is no difficulty to implement it using the declarations `SYMMETRIC` or `ANTISYMMETRIC` of REDUCE. However, most often, indexed objects are neither fully symmetric nor fully antisymmetric: they have *partial* or *mixed* symmetries . In the `DUMMY` package, the declaration `SYMTREE` allows to impose such type of symmetries on operators. This command has been improved and extended to apply to tensors. In order to illustrate it, we shall take the example of the wellknown Riemann tensor in general relativity. Let us remind the reader that this tensor has four indices. It is separately *antisymmetric* with respect to the interchange of the first two indices and with respect to the interchange of the last two indices. It is *symmetric* with respect to the interchange of the first two and the last two indices. In the illustration below, we show how to express this and how `CANONICAL` is able to recognize mixed symmetries:

```
 tensor r; ==> t

 symtree(r,{!+,{!-,1,2},{!-,3,4}});

 rem_dummy_indices a,b,c,d; % free indices

 ra:=r(b,a,c,d); ==>

              b a c d
        ra := r

 canonical ra; ==>

           a b c d
         - r

 ra:=r(c,d,a,b); ==>
                              c d a b
                       ra := r

 canonical ra; ==>

         a b c d
        r


canonical r(-c,-d,a,b); ==>

         a b
        r
            c d

 r(-c,-c,a,b); ==>  0


 ra:=r(-c,-d,c,b); ==>

                  c b
        ra := r
              c d

 canonical ra; ==>

            b c
```

```
          - r
                c d
```

In the last illustration, contravariant indices are placed in front of covariant indices and the contravariant indices are transposed. The superposition of the two partial symmetries gives a minus sign.

There exists an important (though natural) restriction on the use of SYMTREE which is linked to the algorithm itself: Integer used to localize indices must start from 1, be *contiguous* and monotoneously increasing. For instance, one is not allow to introduce

```
symtree(r,{!*,{!+,1,3},{!*,2,4}});

symtree(r,{!*,{!+,1,2},{!*,4,5}};

symtree(r,{!*,{!-,1,3},{!*,2}});
```

but the subsequent declarations are allowed:

```
symtree(r,{!*,{!+,1,2},{!*,3,4}});

symtree(r,{!*,{!+,1,2},{!*,3,4,5}});

symtree(r,{!*,{!-,1,2},{!*,3}});
```

The first declaration endows `r` with a *partial* symmetry with respect to the first two indices.

A side effect of `SYMTREE` is to restrict the number of indices of a generic tensor. For instance, the second declaration in the above illustrations makes `r` depend on 5 indices as illustrated below:

```
symtree(r,{!*,{!+,1,2},{!*,3,4,5}});

canonical r(-b,-a,d,c); ==>

     ***** Index '5' out of range for

            ((minus b) (minus a) d c) in nth

canonical r(-b,-a,d,c,e); ==>

          d c e
    r                    % correct
```

```
           a b

       canonical r(-b,-a,d,c,e,g); ==>

             d c e
          r          % The sixth index is forgotten!
           a b
```

Finally, the function `REMSYM` applied on any tensor identifier removes all symmetry properties.

Another related question is the frequent need to symmetrize a tensor polynomial. To fulfill it, the function `SYMMETRIZE` of the package `ASSIST` has been improved and generalised. For any kernel (which may be either an operator or a tensor) that function generates

- the sum over the cyclic permutations of indices,
- the symetric or antisymetric sums over all permutations of the indices.

Moreover, if it is given a list of indices, it generates a new list which contains sublists wich contain the relevant permutations of these indices

```
       symmetrize(te(x,y,z,{v}),te,cyclicpermlist); ==>


            x y z         y z x         z x y
          te      (v) + te      (v) + te      (v)

       symmetrize(te(x,y),te,permutations); ==>

             x y     y x
            te    + te

       symmetrize(te(x,y),te,permutations,perm_sign); ==>

             x y     y x
            te    - te

       symmetrize(te(y,x),te,permutations,perm_sign); ==>

             x y     y x
           - te    + te
```

If one wants to symmetrise an expression which is not a kernel, one can also use

SYMMETRIZE to obtain the desired result as the next example shows:

```
      ex:=te(a,-b,c)*te1(-a,-d,-e); ==>

                   a   c
             ex := te     *te1
                     b       a d e

      ll:=list(b,c,d,e)$ % the chosen relevant indices

      lls:=symmetrize(ll,list,cyclicpermlist); ==>

         lls := {{b,c,d,e},{c,d,e,b},{d,e,b,c},{e,b,c,d}}

  % The sum over the cyclic permutations is:

   excyc:=for each i in lls sum

                 sub(b=i.1,c=i.2,d=i.3,e=i.4,ex);  ==>


               a   c             a   d
      excyc := te     *te1     + te     *te1
                 b       a d e      c       a e b


                 a   e             a   b
              + te     *te1     + te      *te1
                   d      a b c       e       a c d
```

CANONICAL and tensor derivatives

Only ordinary (partial) derivatives are fully correctly handled by CANONICAL. This is enough, to explicitly construct covariant derivatives. We recognize here that extensions should still be made. The subsequent illustrations show how CANONICAL does indeed manage to find the canonical form and simplify expressions which contain derivatives. Notice, the use of the (modified) DEPEND declaration.

```
      on onespace;

      tensor te,x; ==> t
```

```
    depend te,x;


    aa:=df(te(a,-b),x(-b))-df(te(a,-c),x(-c))$

    canonical aa; ==> 0

    make_partic_tens(eta,eta); ==>  t

    signature 1;

    aa:=df(te(a,-b),x(-b))$

    aa:=aa*eta(-a,-d);

                     a
        aa := df(te     ,x )*eta
                       b   b     a d

  canonical aa; ==>

                 a   a
        df(te      ,x )
               d
```

In the last example, after contraction, the covariant dummy index `b` has been changed into the contravariant dummy index `a`. This is allowed since the space is metric.