

# The Canon package: a fast kernel for tensor manipulators

L.R.U. Manssur and R. Portugal

*Laboratório Nacional de Computação Científica (LNCC), Av. Getúlio Vargas 333,  
Petrópolis, RJ, CEP 25651-075, Brazil*

---

## Abstract

This paper describes the Canon package written in the Maple programming language. Canon's purpose is to work as a kernel for complete Maple tensor packages or any Maple package for manipulating indexed objects obeying generic permutation symmetries and possibly having dummy indices. Canon uses Computational Group Theory algorithms to efficiently simplify or manipulate generic tensor expressions. We describe the main command to access the package, give examples, and estimate typical computation timings.

*Key words:* tensor calculus, Maple, computational group theory, computer algebra  
*PACS:*

---

---

*Email addresses:* leon@lncc.br (L.R.U. Manssur), portugal@lncc.br (R. Portugal).

# LONG WRITE-UP

## 1 Introduction

Abstract tensor manipulation is a desirable feature of any multiple purpose computer algebra system such as Mathematica, Maple, or Mupad. Despite this fact, all available tensor packages are inefficient in manipulating indices obeying permutation symmetries. The calculations with the Riemann tensor polynomials, which have many applications in theoretical Physics, highlight this problem. [1] For example, the product of seven Riemann tensors has 28 indices, and a naïvely implemented tensor package would deal with factorial of 28 index configurations. The exponential number of index configurations requires a careful strategy to address the index manipulation problem.

Fortunately, index canonicalization can be reduced to the problem of finding the canonical representatives of single [2] or double cosets [3] of permutation groups. This reduction is addressed in Ref. [4]. That reference describes an experimental implementation which was used to estimate the algorithm's efficiency. That implementation was improved becoming the Canon package<sup>1</sup>, which is described in the present paper. Canon's purpose is to be a kernel for Maple tensor packages. Its design and implementation is taut, using very basic structures. A small user interface (`CanonPrint`) was included.

A complete tensor (or indexed object) package consists of a large number of tools and built-in tensors (objects) which depend heavily on the application area. The indexed objects can be tensors, but also can be spinors, differential forms, objects with gauge indices and so on. In any case, the indices usually obey permutation symmetries, and there are dummy indices. As mentioned earlier, the manipulation of indices is not trivial, and can be performed by some black box using sophisticated techniques from computational group theory.[5–7] These techniques are not required in designing most of the commands of a complete tensor package. Canon's purpose is to be such black box.

In object oriented languages, objects are the black boxes. What matters is the description of what is the object input and what is the output. The internal structure does not matter. In this work we follow the same strategy. We describe in details the “methods” of the Canon package, which can be loaded by issuing

```
> libname := 'c:/Canon', libname:
> with(Canon);
```

---

<sup>1</sup> The Canon's web page is <http://www.cbpf.br/~portugal/Canon.html>.

[*CanonDefine*, *CanonPrint*, *CanonUnPrint*, *Canonical*, *CanonicalOne*]

These external commands allow a user to communicate with the internal procedures. We do not describe the internal commands except for a brief mention of the main ones in the Appendix. They rely on computational group theory algorithms, such as the one for finding strong generating sets of permutation groups, which are implemented in Canon. Note that there is a group package in Maple, but it does not have the key algorithms needed here. We have re-implemented even basic algorithms such as multiplication of permutations because of efficiency requirements. No command of the Maple group package was used.

This paper is organized as follows: in section 2, we define symmetries and establish the notation to be used; in section 3, we address the problem of algebraic constraints; in section 4, we describe the procedures of the package with some examples; in section 5 we present a performance study; in section 6 we make some remarks on computational complexity and possible applications and finally we draw our conclusions.

## 2 Symmetries

In general, indexed objects obey permutation symmetries, which are commonly described as a set of equations of the form

$$T^{i_1 \dots i_n} = \epsilon_\sigma T^{\sigma(i_1 \dots i_n)} \quad (1)$$

where  $\sigma(i_1 \dots i_n)$  is a permutation of  $i_1 \dots i_n$  and  $\epsilon_\sigma$  is either 1 or  $-1$ . For example, the permutation symmetries of the Riemann tensor are usually described as

$$R^{i_1 i_2 i_3 i_4} = -R^{i_2 i_1 i_3 i_4} \quad (2)$$

$$R^{i_1 i_2 i_3 i_4} = -R^{i_1 i_2 i_4 i_3} \quad (3)$$

$$R^{i_1 i_2 i_3 i_4} = R^{i_3 i_4 i_1 i_2}. \quad (4)$$

From these permutation symmetries, one can derive new ones. In the case of the Riemann tensor, there is a total of 8 equations. The total number of equations appears mysterious in tensor notation, but it is naturally understood in group notation. The permutation symmetries associated with Eqs. (2)-(4) can be expressed by permutations in the group  $\{1, -1\} \otimes S_4$  (the direct product of the multiplicative group  $\{1, -1\}$  and the symmetric group of degree 4): they correspond to  $-(1, 2)$ ,  $-(3, 4)$  and  $(1, 3)(2, 4)$  respectively. These permutations are the generating set of the group of symmetries of the Riemann tensor. The

whole group can be obtained by exhaustive multiplication of the permutations (composition of permutations) of the generating set, yielding

$$S = \{+( ), -(1, 2), -(3, 4), (1, 3)(2, 4), (1, 3)(2, 4), \\ -(1, 3, 2, 4), -(1, 4, 2, 3), (1, 4)(2, 3)\}. \quad (5)$$

The order of the group  $S$  (subgroup of  $\{1, -1\} \otimes S_4$ ) gives the total number of tensor equations. In the present context, it is important to stress that the size of the generating set of a group  $G$  of order  $|G|$  can be at most  $\lfloor \log_2 |G| \rfloor$ . The symmetry group can be large, such as factorial of  $n$  elements for rank- $n$  tensors, but the size of the generating set scales with  $n$ . This feature allows the efficient manipulation of tensors with many indices. [4]

In Canon, a minus-signed permutation is represented by a Maple list, where the first element is  $-1$ , and the second element is a list of lists describing the permutation. For example, symmetries associated with Eqs. (2)-(4) are represented as  $[-1, [[1, 2]]]$ ,  $[-1, [[3, 4]]]$ , and  $[[1, 3], [2, 4]]$ , respectively. The form  $[+1, [[1, 3], [2, 4]]]$  is not allowed. On the other hand,  $[-1, [[1, 3], [2, 4]]]$  is acceptable, which represents the signed permutation  $-(1, 3)(2, 4)$ . The permutations in  $S$  act on the position of the indices of Eqs. (2)-(4) as described in Eq. (1).

### 3 Algebraic Constraints

By now one might be asking about the cyclic symmetry of the Riemann tensor:

$$R^{i_1 i_2 i_3 i_4} + R^{i_1 i_4 i_2 i_3} + R^{i_1 i_3 i_4 i_2} = 0. \quad (6)$$

This equation cannot be represented as a permutation symmetry. It is an algebraic constraint which cannot be addressed by the techniques of computational group theory. To deal with (6) one needs to use group algebra (group ring) algorithms,[8] which address expressions of the type

$$( ) + (234) + (243) = 0. \quad (7)$$

Unfortunately, group algebra algorithms have exponential complexity on the number of indices, associated with the exponential number of Young tableaux [9]. Alternative approaches [10,11] seem to be more convenient, although they lose the generality which is provided by the group algebra approach. An efficient approach (in terms of timing) for simplifying Riemann tensor polynomials is to use a table of rules, which stores all algebraic constraints of Riemann

tensor monomials up to some degree. These constraints are used after the canonicalization as the one described in this work. The table approach is used in Ref. [12]. The Canon package does not address algebraic constraints.

## 4 User commands

### 4.1 *CanonDefine*

The command **CanonDefine** associates a symmetry with a tensor. The syntax is

```
CanonDefine(tensor name, number of indices, generating set,
base)
```

The argument **base** is optional. For example, to define the Riemann tensor, one issues the command

```
> CanonDefine(R, 4, { [-1, [[1, 2]]], [[1, 3], [2, 4]] });
```

$$CanonTable_{R,4_0} = \{[-1, [[1, 2]]], [-1, [[3, 4]]], [[1, 3], [2, 4]], [1, 3]\}$$

Notice that it is enough to give 2 permutations for the symmetries of the Riemann tensor, since the above set generates the group  $S$  of Eq. (5). As an initial step, **CanonDefine** verifies if **generating set** is a strong generating set with respect to the **base**. If the base is not given or the generating set is not strong, a internal procedure generates them using Schreier-Sims' algorithm.[6] If one wishes to use the standard base  $[1, 2, 3, 4]^2$ , one must pass it as the fourth argument of **CanonDefine**. As a final step, **CanonDefine** stores the strong generating set and the associated base in the table **CanonTable**. Notice that the permutation  $-(3, 4)$  was added and the base  $[1, 3]$  was generated in the above example. To verify which tensors are defined and their respective symmetries, one simply prints the table **CanonTable**.

```
> print(CanonTable);
table([(R, 4) = table([0 = ({[-1, [[1, 2]]], [-1, [[3, 4]]], [[1, 3], [2, 4]]}, [1, 3])]))]
```

The tensor symmetry and base are stored in the index 0 of **CanonTable**[R,4]. Then, there is room for other types of data related to the tensor **R** with 4 indices. It is allowed to use tensors with the same name but with different number of indices.

---

<sup>2</sup> Notice that the notations for bases and citations are the same.

## 4.2 Canonical

The command `Canonical` puts a tensor expression into its equivalent normal form. If the input is a single tensor, the output is the canonical form, characterized by the indices being in the least ordering (the definition of permutation ordering is given in [4]). Tensors are indexed objects such as  $T[a, -b]$ , where  $T$  is the tensor name, and positive (negative) indices represent contravariant (covariant) indices. Dummy indices are represented by repeated indices with opposite sign, such as  $T[i, -i]$ . Dummy indices may have symmetries induced by the symmetries of the metric tensor. There are 3 cases: (1) symmetric metric (as usual in tensor calculus),  $T[i, -i] = T[-i, i]$ , (2) anti-symmetric metric (as usual in spinor calculus),  $T[i, -i] = -T[-i, i]$ , and (3) metric with no symmetry (as usual in affine tensor calculus),  $T[i, -i] \neq \pm T[-i, i]$ . Canon uses the global variable `CanonMetricSymmetry := +1, -1` or `0` to specify the 3 cases, respectively. By default it is assigned the `+1` value. For example, the canonicalization of the Riemann tensor is obtained by issuing

```
> Canonical( R[-b, i, -i, a] );
```

$$-R^{a i}{}_{b i}$$

`Canonical` consults the table `CanonTable` to verify the symmetries of the tensors in the input. The next example is more elaborated.

```
> tensor_expression := CanonPrint( R[-e, j, i, b] * R[-b, -c, a, d] *
R[-d, c, -a, e] );
```

$$tensor\_expression := R_e{}^{j i b} R_{b c}{}^{a d} R_d{}^c{}_a{}^e$$

```
> Canonical(tensor_expression);
```

$$-R^{i a j b} R_a{}^{c d e} R_{b d c e}$$

To address the canonicalization of tensor monomials, Canon uses the reduction algorithm described in Ref. [11]. Products of tensors are merged into a single tensor which inherits the symmetries of the original tensors. The merged tensor can be canonicalized using the computational group theory techniques. In the end, the canonicalized merged tensor is converted back to a tensor monomial with the indices of each tensor in the canonical ordering. The output is not canonical with respect to term ordering, since the product is commutative. In this case, we say the output is in a normal form.

### 4.3 CanonicalOne

The command `CanonicalOne` is a shortcut to canonicalize a single tensor. One does not define the tensor, but pass the symmetry in the form of a strong generating set as the second argument and the base as the third argument of `CanonicalOne`. The syntax is

```
CanonicalOne (a single tensor, strong generating set, base).
```

### 4.4 Interface

The interface is controlled by the command `CanonPrint`. It wraps all indexed objects of an expression by the function `_TENSOR`, which displays contravariant (positive) indices in the upper level and covariant (negative) indices in the lower level. For example

```
> CanonPrint( T[a, -b] );  

$$T^a{}_b$$

```

The command `lprint` reveals the hidden function:

```
> lprint( % );  
_TENSOR(T[a,-b])
```

The function `_TENSOR` has an associated procedure `'print/_TENSOR'` which prints positive indices as powers and negative indices as table indices generating the desired display effects in Maple.

After issuing the commands `CanonPrint`, `Canononical`, or `CanonicalOne`, all indexed objects of an expression are wrapped by the function `_TENSOR`. The command `CanonUnPrint` remove the wrapping, and the expression returns to its original form with no interface.

## 5 Performance

The single coset algorithm (free indices) is known to be polynomial, while the double coset algorithm (dummy indices) is known to be exponential in the worst case. On the other hand, the symmetries of tensor expressions are special cases of subgroups of  $\{1, -1\} \otimes S_{2n}$ , and practical calculations show that Canon is efficient enough.

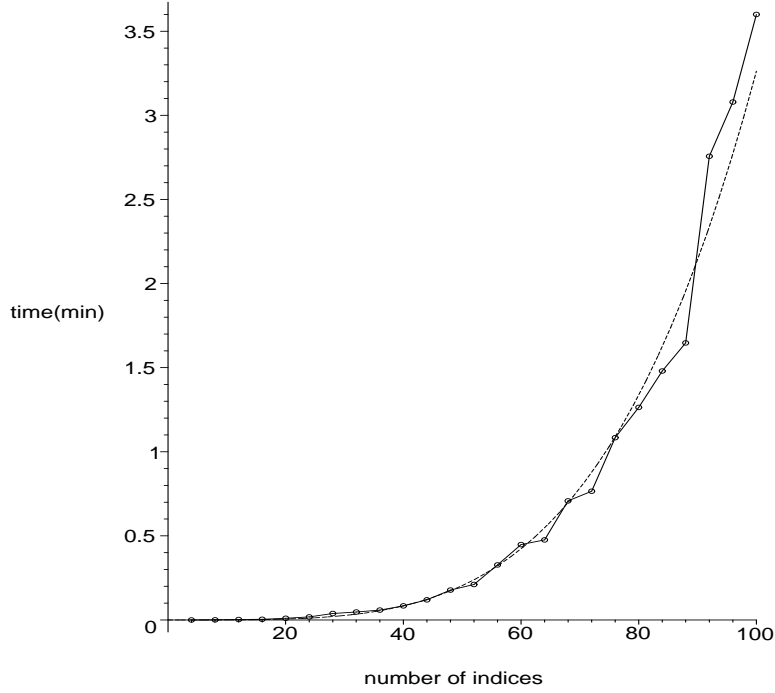


Fig. 1. Timing to find the canonical form of a Riemann monomial versus the number of indices. The dashed line is a fitting curve of the form  $y = 3.3 \times 10^{-8} x^4$ , where  $x, y$  are the horizontal and vertical axis respectively.

The symmetries of the Riemann tensor are some of the most complex that occur in practice. Therefore, monomials built out of Riemann tensors are examples of complex tensor expressions. We have developed an auxiliary program that generates at random Riemann monomials of any degree (number of Riemann tensors) with all indices contracted (Riemann scalar invariants). For each Riemann monomial we calculate the time to find the canonical representative. We have used a 1.2 GHz processor PC, and we have excluded from the count all timings of vanishing results. The vertical axis of the plot of Fig. 1 is the mean value of 40 timings for each monomial. The horizontal axis is the number of indices.

Fig. 1 shows that Canon can handle monomials with a large number of indices. The storage space is very low in order to produce the data. If we try to fit the simulated curve with a polynomial curve of the form  $y = ax^N$ , for integer  $N < 4$  the curve passes above the simulated curve, and for  $N > 4$  the curve passes below for most points. Using the least square method, the best fit is  $y = 3.3 \times 10^{-8} x^4$ . Notice that the deviation from the polynomial curve grows with the degree due to the fact that 40 timings give worse and worse statistics with increasing degree.

Now we give the timing (in seconds) and the allocated memory for verifying that scalar invariants built out of the Riemann tensor with no Ricci tensors



are zero. We give 3 examples of degrees 5, 6, and 7.

```
> byal := kernelopts(bytesalloc):
> invar1 := CanonPrint(R[a,b,c,d]*R[-a,-b,-c,e]*R[-d,-e,f,g]*
R[-f,h,i,j]*R[-g,-h,-i,-j]);
```

$$invar1 := R^{a b c d} R_{a b c}^e R_{d e}^{f g} R_f^{h i j} R_{g h i j}$$

```
> invar2 := CanonPrint(R[a,b,c,d]*R[-a,-b,-c,e]*R[-d,-e,f,g]*
R[-f,-g,h,i]*R[-h,j,k,l]*R[-i,-j,-k,-l]);
```

$$invar2 := R^{a b c d} R_{a b c}^e R_{d e}^{f g} R_f^{h i} R_h^{j k l} R_{i j k l}$$

```
> invar3 := CanonPrint(R[a,b,c,d]*R[-a,-b,-c,e]*R[-d,-e,f,g]*
R[-f,-g,h,i]*R[-h,-i,j,k]*R[-j,l,m,n]*R[-k,-l,-m,-n]);
```

$$invar3 := R^{a b c d} R_{a b c}^e R_{d e}^{f g} R_f^{h i} R_{h i}^{j k} R_j^{l m n} R_{k l m n}$$

```
> showtime( );
> Canonical(invar1);
```

0

```
time = 0.71, bytes = 7149866
> Canonical(invar2);
```

0

```
time = 0.98, bytes = 10165606
> Canonical(invar3);
```

0

```
time = 1.39, bytes = 14129170
> off;
> kernelopts(bytesalloc) - byal;
```

196572

The amount of bytes allocated is less than 200 kB. The total number of non-equivalent zero invariants is: 5 for degree 5, 27 for degree 6, and 248 for degree 7.

## Appendix: Main internal commands

This appendix describes the key internal algorithms of Canon. Most of them are based on computational group theory and are responsible for the package efficiency. All internal commands have names of the form `'Canon/command'`. The total number is above 50 commands and the user can find a brief description of each of them in the accompanying code (Canon.mpl). The arguments between square brackets are optional.

`'Canon/product'( perm1, perm2 )` calculates the product of the two permutations `perm1*perm2` (permutations are applied to the right). Permutations can be minus signed or not.

`'Canon/orbit'( pt, listperms, [ B, sch, back ] )` calculates the orbit of point `pt` in the group generated by the list of (signed) permutations `listperms`. Optionally, if given a basis `B`, and names `sch` and `back`, calculates the Schreier vector and backward pointers.

`'Canon/trace'( pt, sch, back )` traces a Schreier vector. The arguments `sch` and `back` are the Schreier vector and the associated backward pointer, respectively, as assigned by the procedure `'Canon/orbit'`.

`'Canon/SortPermutations'( listperms, B )` sorts the list of permutations `listperms` with respect to base `B`.

`'Canon/singcoset'( perm, strongS, B, free_ind )` obtains the canonical representative of the single coset, which contains the permutation `perm`. The group is generated by the list of strong generators `strongS` with respect to base `B`. `free_ind` specifies the positions to be affected, when using this procedure in connection with `'Canon/doubcoset'`. Output is in the form of a sequence of 3 elements: resulting permutation, updated strong generators, updated free indices positions.

`'Canon/doubcoset'( perm, strongS, BS, strongD, BD )` finds the canonical representative of the double coset, which contains the permutation `perm`. The groups are generated by the lists of strong generators `strongS` with respect to base `BS` and `strongD` with respect to `BD`.

`'Canon/isingroup'( perm, strongS, B)` performs membership testing. It returns `true` if `perm` is in the group generated by the list of strong generators `strongS` with respect to base `B`, and `false` otherwise.

`'Canon/schreier-sims'( K, [ B ] )` uses the Schreier-Sims algorithm [6] in order to find a strong generating set given a generating set `K`, and optionally a base `B`. If not given a base, a base is obtained from `K`.

‘`Canon/indices`’( `expr` ) finds the indices of the tensorial expression `expr`. The output is a list of lists: `[[dummy], [free], [numerical]]`. This command does not use computational group algorithms.

## Acknowledgements

R. P. thanks K. Lake, N. Pelavras and I. Mustapha for the kind hospitality at Queen’s University, where this work was started. R. P. was partially supported by CNPq and L.R.U.M. was supported by CNPq and PCI/MCT.

## References

- [1] S.A. Fulling, R.C. King, B.G. Wybourne and C.J. Cummins, *Normal forms for tensor polynomials: I. The Riemann tensor*, Class. Quantum Grav. **9** (1992) 1151-1197.
- [2] G. Butler, *Effective Computation with Group Homomorphisms*, J. Symbolic Comp. **1** (1985) 143-157.
- [3] G. Butler, *On Computing Double Coset Representatives in Permutation Groups*, Computational Group Theory, ed. M.D. Atkinson, Academic Press (1984) pp. 283-290.
- [4] L.R.U. Manssur, R. Portugal, B.F. Svaiter, *Group-theoretic Approach for Symbolic Tensor Manipulation*, Int. J. Mod. Phys. C **13** (2002) 859-880.
- [5] C.C. Sims, *Computation with Permutation Groups*, Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation (Los Angeles 1971), ed. S.R. Petrick, ACM, New York (1971).
- [6] J.S. Leon, *On an Algorithm for Finding a Base and a Strong Generating Set for a Group Given by Generating Permutations*, Math. Comp. **35** (1980) 941-974.
- [7] G. Butler, *Fundamental Algorithms for Permutation Groups*, Lecture Notes in Computer Science, vol. 559, Springer-Verlag (1991).
- [8] B. Fiedler, *A Use of Ideal Decomposition on the Computer Algebra of Tensor Expressions*, Zeitschrift für Analysis und ihre Anwendungen **16** (1997) 145-164.
- [9] H. Boerner, *Representations of Groups*, North-Holland Pub. Co., Amsterdam (1970).
- [10] A. Balfagon, X. Jaen, *Review of some classical gravitational superenergy tensors using computational techniques* Class. Quantum Grav. **17** (2000) 2491-2497.

- [11] R. Portugal, *Algorithmic Simplification of Tensor Expressions*, J. Phys. A: Math. Gen. **32** (1999) 7779-7789.
- [12] L. Parker, S.M. Christiansen, *MathTensor, A system for Doing Tensor Analysis by Computer*, Addison-Wesley, Reading, MT, 1994.