Tensor computer algebra with Redberry

D. Bolotin, S. Poslavsky
(IBCH RAS Moscow, IHEP Protvino, Russia)

`bolotin.dmitriy@gmail.com`
`stvlpos@mail.ru`

Introduction

Computer algebra (CA) techniques are now widely used in scientific research. In the area of theoretical physics and particularly high-energy physics, the standard mathematical formalism is largely based on tensor algebra and substantial part of analytical calculations consist of algebraic manipulations with abstract tensors (or more generally, objects with indices). From the standpoint of CA, the main distinctive feature of tensorial expressions arises from the presence of dummy indices (summation indices): contractions between these indices form a graph of multipliers, instead of a simple list of multipliers in case of ordinary non-tensorial expressions. For illustration consider the following tensors, where the Einstein convention holds (implied summation over repeated indices) and for the sake of simplicity we assume that both T_{abcd} and T_{abc} are fully symmetric and don't distinguish co- and contravariant indices:

$$T_{abmj}\,T_{defi}\,T_{abc}\,T_{cdj}\,T_{inm}\,T_{efn} \qquad (11a)$$
$$T_{mnij}\,T_{ebfa}\,T_{mic}\,T_{adb}\,T_{dne}\,T_{fcj} \qquad (11b)$$
$$T_{abmj}\,T_{defi}\,T_{abc}\,T_{cef}\,T_{inm}\,T_{djn} \qquad (11c)$$

One can prove that tensors (11a) and (11b) are equal while (11c) is different. This can be directly observed from the graph representation of the expressions (Fig. 4): each multiplier T corresponds to a graph vertex and index contraction between two T's to a graph edge between the corresponding vertices.

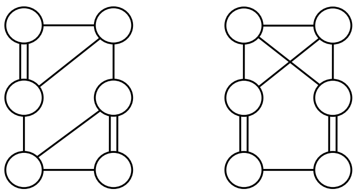

Figure 4: *Graph representation of expressions* (11a), (11b) *(left) and* (11c) *(right). Vertices correspond to multipliers T and edges (lines) indicate index contractions.*

From this perspective it is clear that testing whether two tensorial expressions are equal is in general at least as hard as testing whether corresponding graphs are isomorphic (graph isomorphism (GI) problem).

In most practical cases in physics we rarely deal with completely symmetric tensors: usually, tensors have sophisticated permutational symmetries or do not have symmetries at all. In the latter case the problem of tensor comparison (matching) can be significantly simplified: each dummy index has a distinctive property – an unordered pair (μ, ν) where μ, ν are the positions of the dummy index within the indices of the multipliers (for example, $(1, 3)$ for index a in product $T_{abc}\,T_{cba}$), so by comparing sets of such pairs we can test whether two expressions can be matched within polynomial time.

On the other hand, in case of nontrivial symmetries the problem of tensor matching becomes even more complicated than the GI problem. For example, if tensors from the first example are not fully symmetric but have more specialised symmetries, e.g. $T_{abcd} = T_{bacd} = T_{cdab}$ (forms a permutation group with order 8) and $T_{abc} = T_{bca}$ (the simple alternating group) then (11a) becomes unequal to (11b), while the corresponding graphs are still isomorphic.

In the general case a tensor has symmetries forming a permutation group G. In comparing products of such tensors, suppose we found a possible match between two vertices and denote the corresponding set of possible matches of their edges as permutation α. Then, in addition to GI, we also have to test that $\alpha \in G$ (membership testing). While membership testing can be done in polynomial time (see e.g. Sect. 4.4 in [1] for details), there is still no known polynomial-time algorithm for the GI problem.

Summarising, we see that even such a basic operation as *atomic comparison* (i.e., testing equality without using any specific transformations or simplifications) of tensorial expressions is very complicated, which drastically complicates implementation of standard CA transformations like e.g. reduction of similar terms, which is straightforward in the case of non-tensorial expressions.

Today there is a wide range of packages (e.g. xAct[2] for Mathematica or Maple's Physics package) and standalone tools (e.g. Cadabra [3, 4]) that cover different topics in symbolic tensor calculus. As far as we know, all open-source packages are based on the so-called *index canonicalisation* approach for handling and simplifying tensorial expressions. The idea of this approach is widely used in scalar CASs and consists of putting each expression into a unique canonical form (thereby making comparison a trivial operation). In the case of scalars this is generally achieved by sorting of monomials (e.g. lexicographically), while in the case of tensors it is complicated by the presence of symmetries and contractions. If '$\prec$' denotes (e.g.) lexicographical ordering on the set of all indices, the canonical form of an expression (which can be obtained by reordering multipliers, renaming of dummies, and permuting indices using symmetries defined for tensors) can be defined as the $\prec$-least configuration of indices written in the order in which they appear in the expression.

Canonical forms for the expressions from the first example are:

$$T_{ade}\,T_{abc}\,T_{fmn}\,T_{fij}\,T_{bcdi}\,T_{ejmn} \qquad (12a)$$
$$T_{ade}\,T_{abc}\,T_{fmn}\,T_{fij}\,T_{bcim}\,T_{dejn} \qquad (12b)$$

where (12a) is a canonical form of (11a), (11b) and (12b) of (11c). The corresponding index configurations

$$S_1 = (a, d, e, a, b, c, f, m, n, f, i, j, b, c, d, i, e, j, m, n),$$
$$S_2 = (a, d, e, a, b, c, f, m, n, f, i, j, b, c, i, m, d, e, j, n)$$

are the $\prec$-least of all permutations of indices that can be obtained by shuffling multipliers, renaming dummies and permuting indices using the symmetries of tensors T_{abcd} and T_{abc}.

It can be shown [5, 6] that the problem of tensor canonicalisation is equivalent to the problem of finding canonical representatives of double cosets in permutation groups, which is $\mathcal{NP}$-hard in general. Another important drawback of the approach is that it is applicable only to the products of simple tensors, so if e.g. a product contains a sum as one of its multipliers, expanding out the brackets is required for canonicalisation of the expression. Finally, the canonical form depends on the particular names of the indices, which renders it unsuitable in conjunction with substitutions involving summation indices: even if both, substitution rule and target, are written in canonical form, they may have completely different order of indices and multipliers. For example, before $T_{abc}T_{eab} = F_{ce}$ can be substituted into $T_{abp}T_{acq}T_{bpc}$, one still has to match the left-hand side in target to yield $F_{ca}T_{acq}$, which is not trivial even though all expressions are written in canonical form (in the presence of symmetries, this becomes even more complicated).

Here we discuss the computer algebra system Redberry [7] which is focused on algebraic manipulations with tensors and uses a graph-based approach for handling such expressions. From a graph-theoretical point of view a problem of comparing two tensorial expressions is equivalent to the GI problem, and e.g. substitution (matching a subexpression in the target expression) is equivalent to a subgraph isomorphism problem. The GI problem also has exponential complexity in the worst case but very efficient algorithms are known [8, 9] which work in polynomial time for all practical cases.

Besides a new approach, one of the main motivations in the development of a new system was to provide abilities for the user to implement custom functionality; we started development six years ago when we failed to find an appropriate customizable tool for our research in gravity. Cadabra was a perfect candidate but it does not provide a programming language for implementing even simple user routines (looping, if–else, functions, etc.).

Redberry provides a basic computer algebra toolset (algebraic manipulations, substitutions, basic simplifications etc.) as well as tools for calculations in high-energy physics: Dirac and SU(N) algebra, Levi-Civita simplifications and one-loop counterterm calculation in quantum field theory. Redberry is written in Java; for the user interface we implemented a simple domain-specific language in Groovy. Comprehensive details on Redberry installation and usage can be found in [7] and on the Redberry website. Here we are going to focus only on CA aspects.

Working in Redberry

Basic examples

Consider a "toy" example to highlight the basics:

```
// define symmetries of Riemann tensor
addSymmetry 'R_abcd', -[[0, 1]].p
addSymmetry 'R_abcd',  [[0, 2], [1, 3]].p
// input tensorial expression
expr = 'R_ibcd*R^bcd_j - R_dcbi*R_j^dcb'.t
// print expression
println expr

  ▷ 0
```

In the first two lines we specify symmetries of the Riemann tensor ($R_{abcd} = -R_{bacd} = R_{cdab}$) using permutations written in disjoint-cycle notation (minus in the first line indicates antisymmetry); the `.p` converts a list of integers into internal representation of permutation and `.t` parses the preceding string expression into the internal tensor representation (as one can see, Redberry distinguishes covariant and contravariant indices and standard LaTeX curly braces are not required for inputting tensor indices). Redberry automatically detects that the two terms in `expr` are equal and reduces the sum.

Similar to many scalar CASs (and in contrast to the majority of tensor software), Redberry puts *any* intermediate and resulting expression into some standard form (SF) (for instance, reduces similar terms in sums), which drastically improves the overall performance of huge calculations. Consider another example:

```
expr='(A_abc-A_bac)*T^cb + (A_iaj-A_aij)*T^ji'.t
println expr

  ▷ 0
```

Here we see that Redberry automatically matched dummy indices and figured out that $(A_{abc} - A_{bac})$ is antisymmetric with respect to a, b. The graph algorithms used in Redberry make reduction to SF extremely fast and light-weight, so one need not take into account performance of such a seemingly complicated simplification.

Substitutions and transformations

Let's turn to substitutions and definition of functions. Consider a simple substitution $R_{ab} = R^{m}{}_{amb}$ in $R_{ba}R^{am}$:

```
subs = 'R_ab = R^m_amb'.t
expr = subs >> 'R_ba*R^am'.t
println expr

  ▷ R^d_bda*R^ca_c^m
```

A rather tricky index relabelling and matching was done automatically to obtain the correct result $R^{d}{}_{bda}R^{ca}{}_{c}{}^{m}$.

Redberry allows to perform substitutions with any kind of left-hand side (sums, products etc.) of any complexity, taking into account index symmetries and contractions (and even symmetries that arise from the structure of expressions):

```
addSymmetry 'R_abc', -[[0, 2]].p
subs = 'R_abc*(R^ba_d - R^ab_d) = F_cd'.t
expr = subs >> 'R_cab*F^c_d*(R^dab + R^abd)'.t
println expr

  ▷ -F^cd*F_cd
```

Redberry admits tensorial functions depending on tensorial arguments as left-hand side of substitutions (performs matching of function arguments):

```
s = 'F_ij[x_m, y_m] = x_i*y_j'.t
t = 'T^ab*F_ab[p^a - q^a, p^a + q^a]'.t
println s >> t

  ▷ T^ab*(p_a-q_a)*(p_b+q_b)
```

As one could already observe, substitutions are applied using the right shift >> operator. This notation is also valid for all other types of transformations, such as in the following code which simplifies an expression containing a metric tensor:

```
tr = Expand & EliminateMetrics & 'd^a_a = D'.t
expr = 'g^mn*g^ab*g^gd*(p_g*g_ba + p_a*g_bg)
        *(p_m*g_dn + p_n*g_dm)'.t
println tr >> expr

  ▷ 2*(1+D)*p^d*p_d
```

The "and" operator `&` used in the first line joins transformations into a single one that sequentially expands brackets, eliminates contracted metrics and substitutes the space dimension D. Built-in notations for the metric `g_ab` and Kronecker delta `d^a_b` are used.

Redberry has dozens of built-in transformations both tensor-specific (symmetrization, simplification of metrics, tensor differentiation etc.) and general-purpose (polynomial factorization etc.) that are specifically adapted to tensor algebra.

High-energy physics features

Redberry was originally written for computations in high-energy physics and gravity. For this, Redberry provides tools for dealing with noncommutative objects like Dirac or SU(N) matrices. A trace of Dirac matrices is calculated e.g. as follows:

```
defineMatrices 'G_a', 'G5', Matrix1.matrix
dTrace = DiracTrace[[Gamma: 'G_a', Gamma5: 'G5']]
expr = 'Tr[p^a*p^b*G_a*G_b*G_c*G_d*(1+G5)]'.t
println dTrace >> expr

  ▷ 4*p^b*p_b*g_cd
```

First, we tell Redberry that tensors `G_a` and `G5` (Dirac's γ_μ and γ_5) are noncommuting matrices (one can also define e.g. spinors) of type `Matrix1` (the type is required to distinguish e.g. Dirac and SU(N) matrices). Then we define the trace transformation in the specified notation for Dirac matrices and apply it to the expression.

Another interesting out-of-the-box feature is the ability to solve equations with tensors. Computing the sum over polarisations for a spin-2 particle requires solution of the following system of equations:

$$J_{abcd}\,p^a = 0, \quad J_{abc}{}^{c} = 0, \quad J_{abcd}J^{abcd} = 5,$$

where J is an unknown tensor with symmetries $J_{abcd} = J_{bacd} = J_{cdab}$ and p is the momentum of the particle (with $p_a p^a = m^2$, where m is its mass). In Redberry one can do:

```
addSymmetries 'J_abcd', [[0,1]].p,[[0,2],[1,3]].p
eq1 = 'J_abcd * p^a = 0'.t
eq2 = 'J_abc^c = 0'.t
eq3 = 'J_abcd * J^abcd = 5'.t
rules = 'd^n_n = 4'.t & 'p_a*p^a = m**2'.t
opts = [Transformations: rules,
        ExternalSolver : [
            Solver: 'Mathematica',
            Path  : '/usr/bin']]

result = Reduce([eq1,eq2,eq3], ['J_abcd'], opts)

  ▷ [[J_{abcd} = (1/2)*m**(-2)*p_{a}*p_{d}*g_{bc
       ..........
```

The result can further be simplified using the relation $J_{ab} = -g_{ab} + p_a p_b/m^2$:

```
println(( 'g_ab = -J_ab+p_a*p_b/m**2'.t
        & ExpandAndEliminate) >> result)
```

which gives the final well-known solution

$$J_{abcd} = \pm\left((J_{ac}J_{bd} + J_{ad}J_{bc})/2 - J_{ab}J_{cd}/3\right)$$

The `Reduce` function converts tensorial equations into equations with pure scalars (by representing unknown tensors in the most general decomposition allowed by the symmetries, e.g. $J_{abcd} = c_0\, g_{ab}g_{cd} + c_1\, p_a p_b p_c p_d + \dots$) and passes this scalar system to an external routine if specified (Mathematica or Maple) or just returns it as is. One can specify additional assumptions for `Reduce`, in our case the space-time dimension was 4 and $p_a p^a = m^2$. Note that the result will have exactly the same symmetries as specified for the unknown variables (first line of the code); in our example this is crucial since without the first `addSymmetries` we would obtain additional solutions.

Readers interested in seeing more examples are referred to [7]. Also, many examples including Feynman-diagram calculations, obtaining Feynman rules from the Lagrangian, calculating one-loop counterterms in general field theory etc. are available on Redberry website.

Under the hood

Symmetries and permutation groups

For handling symmetries of tensors, Redberry implements algorithms for working with permutation groups. These algorithms are also closely related to graph algorithms for comparing expressions (graph isomorphism problem) and finding of symmetries of tensors (graph automorphism problem). The implementation uses bases and strong generating sets (see e.g. Sect. 4.4 in [1]) and provides methods for coset enumeration, searching for centralizers, stabilizers, etc.:

```
// define a Permutation group from generators
gr = Group([[0,1],[4,5]].p, [[3,4,5],[1,2]].p)
// get order of group
println gr.order()
```

```
  ▷ 36
```

```
// find setwise stabiliser of {2,3,5}
println gr.setwiseStabilizer(2, 3, 5)
```

```
  ▷ Group( +[[0, 1]], +[[3, 5]] )
```

```
// define some other PermutationGroup
oth = Group([[2,3],[1,4]].p,[[3,4,5]].p)
// find intersection of groups
intrs = gr.intersection(oth)
println intrs
```

```
  ▷ Group( +[[3, 4, 5]], +[[1, 2], [4, 5]] )
```

The permutation-group package in Redberry implements the most complete set of algorithms and data structures available for such problems as open source in Java; more details can be found on the Redberry website.

Mappings

The central entity used internally in Redberry is a *mapping of indices* between two tensors, i.e. a set of rules on how to rename free indices of one tensor to obtain another (e.g. $\{a \to c, b \to d\}$ for tensors T_{ab} and T_{cd}). Mapping is a result of expression matching (graph isomorphism testing in the case of products) and Redberry uses these mappings for comparison, substitutions, and many other routines. Consider the two monomials

$$R_{pq}{}^{ab}\, R_{abcd}\, R^{cjpq} \quad \text{and} \quad R_{bc}{}^{qp}\, R_{iqap}\, R^{jicb},$$

where $R_{abcd} = R_{cdab} = -R_{dacb}$. One can check that if indices are renamed as $\{d \to a\}$ or $\{d \to j, j \to a\}$ in the first tensor, the result is minus the second tensor (to within dummy-index relabelling and shuffling of indices according to symmetries). In Redberry this can be found out as follows:

```
addSymmetry 'R_abcd', [[0,2],[1,3]].p
addSymmetry 'R_abcd', -[[0,1,3,2]].p
lhs = 'R_abcd*R_pq^ab*R^cjpq'.t
rhs = 'R_iqap*R_bc^qp*R^jicb'.t
mappings = lhs % rhs
mappings.each { m -> println m }
```

```
  ▷ -{_d->_a, _j->_j}
  ▷ -{_d->^j, _j->^a}
```

The object `mappings` constructed from two input tensors using the `%` operator allows to iterate over all possible mappings from `lhs` onto `rhs`. The minus in the output indicates that one needs to negate the result after relabelling `lhs` in order to obtain `rhs`.

Programming capabilities

One of the main motivations for developing Redberry was to provide ability for the users to implement additional functionality; for this purpose Redberry provides a wide range of specialized methods (besides basic things like looping). Consider implementation of a simple substitution using mappings and tree traversal:

```
subs = { expr, rule ->
  expr.transformParentAfterChild { node ->
    //map l.h.s. onto current node
    mapping = rule[0] % node
    mapping.exists ? mapping >> rule[1] : node
  }
}
rule = 'T_ab = A_ai*B^i_b'.t
expr = 'k^j*T_ij*(2*T^ia + T^ai)'.t
println subs(expr, rule)
```

```
  ▷ k^j*B^c_j*A_ic*(B^bi*A^a_b+2*B^ba*A^i_b)
```

The method `transformParentAfterChild` traverses the expression tree (from bottom to top) and applies the specified function to each node. For the `rule` argument we test whether one can map its left-hand side to the current node and if the answer is yes, we replace it with the right-hand side in which we rename indices according to the obtained mapping (again using the >> operator). Note that the appropriate relabelling of dummy indices is performed automatically and the user doesn't need to care about possible clashes of dummy indices.

Performance

It is interesting to compare the performance of Redberry with other tools in the field, especially because, as far as we know, other packages use the index-canonicalisation approach discussed in the Introduction, while Redberry is based on graph algorithms. To compare performance we generated sums of 200 terms of different complexity (of the form $T_{abc}F_{bcad}\ldots$ with varying number of multipliers) and measured the time needed to reduce similar terms in such sums. We studied the dependence of time on the number of multipliers (Fig. 5) and on the total number of indices in products (Fig. 6).

As can be seen from Figs. 5 and 6, Redberry outperforms the other systems in all cases except situations with large number of symmetric tensors with many indices (the reason of this performance degradation is already investigated and will be fixed in an upcoming release).

Further development

Redberry has been actively developed for six years and the source code contains more than 130k lines already; it is covered by more than 1000 unit tests and many real-world computations in physics were performed using Redberry.

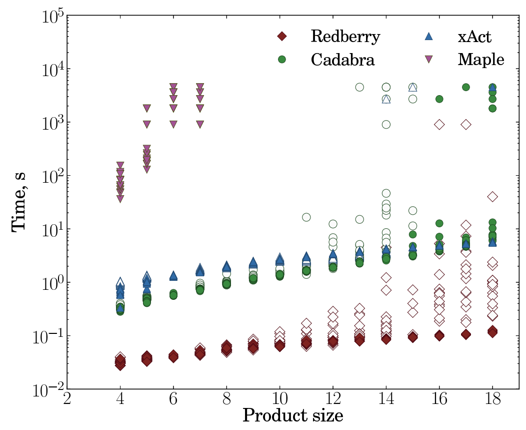

Figure 5: *Performance of different systems for increasing number of multipliers. Filled symbols: no symmetries specified for tensors. Open symbols: all tensors symmetric or antisymmetric (Maple 18 is able to simplify only very trivial tensors with symmetries, so we excluded Maple in this case).*

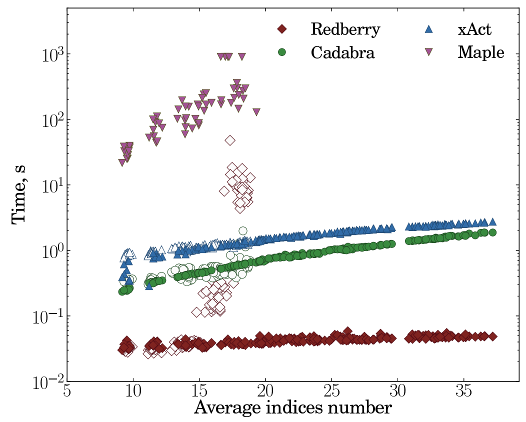

Figure 6: *Performance of different systems for increasing total number of indices. Legend as for Fig. 5.*

Currently we are working on the improvement of graph algorithms, implementation of a complete pattern matching (like in scalar CASs) and improvement of overall stability and usability of the system. The high-energy physics package is frequently updated with new tools for real computations.

Redberry is an open-source package and licensed under GNU GPL v3. The source code repository is at http://bitbucket.org/redberry/redberry and the issue tracker at http://youtrack.redberry.cc.

Comprehensive documentation with lots of examples can be found on http://redberry.cc.

References

[1] D. Holt, B. Eick, E. O'Brien, Handbook of Computational Group Theory. *Discrete Mathematics and Its Applications, Taylor & Francis, 2005, ISBN:1-58488-372-3.*

[2] J. M. Martin-Garcia, xPerm: fast index canonicalization for tensor computer algebra, *Comput.Phys.Commun. 179 (8) (2008) 597 – 603 (arXiv:cs/0803.0862)*, http://www.xact.es

[3] K. Peeters, Symbolic field theory with Cadabra. *Computeralgebra-Rundbrief*, 41:16–19, Oktober 2007.

[4] K. Peeters, A Field-theory motivated approach to symbolic computer algebra, *Comput. Phys. Commun. 176 (2007) 550*, [cs/0608005 [cs.SC]]

[5] A. Ya. Rodionov, A. Yu. Taranov, Combinatorial aspects of simplification of algebraic expressions. *Eurocal'87 Lecture Notes in Computer Science 378 (1989) 192-201.*

[6] L. R. U. Manssur, R. Portugal, B. F. Svaiter, Group-theoretic approach for symbolic tensor manipulation *International Journal of Modern Physics C 13 (07) (2002) 859–879*, arXiv:math-ph/0107032

[7] D. A. Bolotin, S. V. Poslavsky, Introduction to Redberry: a computer algebra system designed for tensor manipulation. *arXiv:1302.1219 [cs.SC]*, February 2013, http://redberry.cc

[8] B. D. McKay, Practical graph isomorphism. *Proceedings of 10th. Manitoba Conference on Numerical Mathematics and Computing (Winnipeg, 1980); Congressus Numerantium 30 (1981) 4587.*

[9] B. D. McKay, A. Piperno, Practical graph isomorphism, II. *Journal of Symbolic Computation 60 (2014) pp. 94-112*, arXiv:cs/1301.1493.