Introduction to Redberry: the computer algebra system designed for tensor manipulation

D A Bolotin  
Institute of Bioorganic Chemistry of the Russian Academy of Sciences, 117997 Moscow, Russia  
E-mail: bolotin.dmitriy@gmail.com

S V Poslavsky  
Institute for High Energy Physics, 142281 Protvino, Moscow Region, Russia  
Department of Theoretical Physics, Moscow State University, 119991 Moscow, Russia  
E-mail: stvlpos@mail.ru

Abstract: In this paper we introduce Redberry — an open source computer algebra system designed to manipulate with symbolic tensorial expressions. It implements basic computer algebra system routines as well as complex tools for real computations in physics. Redberry core provides common for majority of computer algebra systems tools for expressions manipulation, generalized on tensorial objects, as well as tensor-specific features: indices symmetries, LaTeX-style input, natural dummy indices handling, multiple index types etc. The high energy physics package includes tools for Feynman diagrams calculation: Dirac and SU(N) traces, Levi-Civita simplifications and tools for one-loop calculations in general field theory. In the present paper we give detailed description of Redberry functionality: from basic manipulations with tensors to real Feynman diagrams calculation, accompanied by many examples. We also introduce graph representation of a tensor — the basic underlying idea of the Redberry architecture, which clarifies a deep connection between symbolic tensor algebra and computational graph theory. Redberry is written in Java 7 and provides convenient Groovy-based user interface and extensive API for developers. The source code of Redberry is covered by more then 800 tests. Redberry is available from redberry.cc.
# Table of contents

1 Introduction 3  
  1.1 Background ................................................. 3  
  1.2 The introductory example ................................... 3  
  1.3 Paper structure ............................................ 5  

2 Basics 5  
  2.1 Basic input/output .......................................... 5  
  2.2 Tensors ..................................................... 7  
  2.3 Einstein notation ........................................... 8  
  2.4 Standard form of mathematical expressions .................. 9  
  2.5 Indices .................................................... 11  
  2.6 Symmetries of tensors ...................................... 13  

3 Transformations 15  
  3.1 Applying and manipulating transformations .................. 15  
  3.2 Selected general-purpose transformations .................... 16  
    3.2.1 Substitutions .......................................... 16  
    3.2.2 Differentiate .......................................... 19  
    3.2.3 EliminateMetrics ....................................... 20  
    3.2.4 Expand ............................................... 20  
    3.2.5 Factor ................................................. 21  
    3.2.6 Together .............................................. 22  
  3.3 Selected physical transformations ............................. 22  
    3.3.1 DiracTrace ............................................. 22  
    3.3.2 UnitaryTrace .......................................... 23  
    3.3.3 LeviCivitaSimplify .................................... 24  

4 Selected physical applications 25  
  4.1 Feynman diagrams .......................................... 25  
  4.2 One-loop counterterms of arbitrary Lagrangians .............. 29  
    4.2.1 Non-minimal vector field ............................... 30  
    4.2.2 Minimal fourth order operator ......................... 32  

5 Basic internal architecture 34  
  5.1 Mappings of indices ........................................ 34  
    5.1.1 Multiple mappings and symmetries of tensors ........... 35  
    5.1.2 Sign of mapping ...................................... 35  
    5.1.3 Mappings and substitutions ............................ 36  
    5.1.4 Examples and performance ............................. 36  
  5.2 Graph theory and tensors ................................... 37  
  5.3 Expression-tree traversal and modification .................. 40  

6 Conclusions 42  
A Inputting matrix expressions 43  
B Redberry artifacts structure 44
1 Introduction

1.1 Background

General-purpose computer algebra systems (CASs) have become an essential part of many scientific calculations. At the moment there is no CAS that provides generic tools for calculations both with ordinary symbolic expressions and tensors, and at the same time have general programming capabilities. Among the existing standalone systems we would like to emphasize Cadabra developed by Kasper Peeters [1, 2]. In our opinion this CAS provides the most comprehensive set of tensor-oriented tools: tensor symmetries, multiple index types, dummy indices handling, substitutions, comparison of tensorial expressions and tensor-specific transformations like contractions with metrics, symmetrization, simplification of tensorial expressions etc. On the other hand, Cadabra does not provide general programming environment, i.e. such basic constructions as loops, branching and low-level expression manipulation. This makes it impossible to develop user-defined algorithms and modules in Cadabra\(^1\). Also, there is a number of software packages which deal with tensors, built on top of general-purpose CASs and aimed at solving particular problems in physics and mathematics. Among these packages we would like to highlight FeynCalc [3], which provides facilities for calculations in high energy physics (in particular for Feynman graphs). The main drawback of such software packages is that they consider tensors as additional but not a fundamental entities, which restricts the scope of their application by the particular set of problems for which they are intended. Authors have no purpose to make a review of all tensor-oriented software here but only to emphasize the most relevant works.

The main challenge of Redberry is to develop the computer algebra system with native support of tensorial objects, i.e. to generalize basic routines of computer algebra systems onto tensors. Such an approach leads to a fundamentally different point of view on the architecture of the CAS, because tensorial expressions have principally different structure and manipulation rules in contrast to indexless expressions. Redberry is developed in such a way that its functionality is available through the modern high-level general-purpose programming language with excellent IDE support. Thereby, Redberry gives a common language for both particular calculations and development of new modules and algorithms. As an example of the successful use of such a model, one can mention SymPy [4] computer algebra system, which uses Python as a main user interface.

The key features of Redberry include: support of permutational symmetries of tensor indices, extensive tools for comparison of tensorial expressions, native dummy indices handling (including automatic clash resolution), multiple index types, \LaTeX-style input/output and a comprehensive set of tensor-specific transformations. On top of core functionality, Redberry implements a number of tools for calculations in high-energy physics including calculation of Feynman diagrams and one-loop counterterms in general field theory.

Redberry core is written in Java, while the user interface is written in Groovy and is intended to be used within the Groovy environment. Redberry does not provide GUI by itself, but modern open source IDEs\(^2\) give a very convenient way of Redberry usage, including syntax highlighting and code completion. Installation instructions and additional documentation can be found on the Redberry website: http://redberry.cc.

1.2 The introductory example

Let's start from the minimal but still complex real example to quickly dive into Redberry (the next section covers every aspect in details). The example demonstrates calculation of differential cross section of the Compton scattering in scalar QED:

\(^1\)Though it is available on the C++ level of Cadabra.
\(^2\)we recommend JetBrains IntelliJ IDEA because of its brilliant support of Groovy
Listing 1: Compton scattering in scalar QED

```python
def V1 = 'V_i[p_a, q_b] = -I*e*(p_i+q_i)'.
def V2 = 'V_{ij} = 2*I*e**2*g_{ij}'.
def P = 'D[k_a] = -I/(k^a*k_a-m**2)'.
def M = ('M^ij = V^i[p1_a,p1_a+k1_a]*D[p1_a+k1_a]*V^j[-p2_a,-p1_a-k1_a]'
    + 'V^i[p1_a,p1_a-k2_a]*D[p1_a-k2_a]*V^j[-p1_a+k2_a,-p2_a]
    + 'V^ij').t.
M = (V1 & V2 & P) >> M
M2 = M >> 'M2 = -M_ij*M^ij'.t
M2 = ('d_i^i = 4'.t >> M2
M2 = mandelstamSubstitutions >> M2
M2 = 'u = 2*m**2-s-t'.t >> M2
println M2
```

This script will print the well-known expression for the squared matrix element of the Compton scattering in scalar QED:

$$|M|^2 = \frac{e^4}{(s-m^2)^2(-s-t+m^2)^2} \left( m^8 - 4m^6s + 6m^4s^2 - 4m^2s^3 + 
+ s^4 + 2m^4se - 4m^2s^2t + 2s^3t + m^4t^2 + s^2t^2 \right) ,$$

which becomes equal to the differential cross section $d\sigma/d\Omega$ after multiplying by $1/(64\pi^2s)$.

The code above speaks for itself, so leaving aside the physical aspects of the problem, let’s focus only on programming aspects. ‘...’.t construct converts a string representation into a computer object. The input notation for tensors is the same as used in \LaTeX{} with minor syntax modifications\(^1\)(see Sec. 2.1). Redberry uses the predefined notation for some built-in tensors, such as $I$ for image one, or $g_{ij}$ and $d^i_j$ for the metric tensor and Kronecker delta respectively. Thus, for example, when we perform contractions with metrics and deltas in the line 19, Redberry automatically takes into account that $g_{ai} g^{aj} = \delta^j_i$. However, since the dimension of the space affects Kronecker trace value and it is not specified anywhere explicitly, we substitute the trace manually (on line 20).

As could be seen from the example, state of index (upper or lower, i.e. contravariant or covariant) is important — indices are considered to be contracted if they have different states. All transformations in Redberry (e.g. substitutions, Expand, EliminateMetrics etc.) are first-class objects and can be assigned to variables (like in line 22). They can be applied to

\(^1\)One can omit curly braces where it does not cause ambiguity.
mathematical expression using >> operator. Detailed discussion on Redberry transformations usage and list of basic built-in transformations can be found in Sec. 3.

The last point that should be discussed here is setMandelstam(...) function, which is used in the lines 22–23. It takes Lorentz four-vectors with the corresponding particle masses and simply returns a list of substitutions following from provided values. This function is a part of redberry-physics (see Appx. B).

1.3 Paper structure

The whole paper is accompanied by many Groovy examples. Each example can be executed by simple wrapping the code in the following way:

```
@Grab(group = 'cc.redberry', module = 'groovy', version = '1.1')
import cc.redberry.groovy.Redberry
import static cc.redberry.core.tensor.Tensors.*
import static cc.redberry.groovy.RedberryPhysics.*
import static cc.redberry.groovy.RedberryStatic.*

use(Redberry){
    //example code
}
```

This will automatically download Redberry and all required dependencies. Some examples also may require additional import statements, in this case it will be noted in a footnotes.

In Sec. 2 we give description of Redberry syntax and basic functionality, needed for usage and understanding of main Redberry principals. In Sec. 3 one can find a list of selected transformations and general aspects concerning their application. Sec. 4 illustrates Redberry application in high energy physics: complete calculation of Compton scattering in QED and complete calculation of one-loop counterterms of non-minimal vector field and minimal fourth order operator. In Sec. 5 we consider selected architectural solutions used in Redberry including graph representation of tensors.

2 Basics

2.1 Basic input/output

Let’s start detailed description of Redberry from discussion on syntax used to define tensorial expressions. As mentioned above, there is a special syntax construction to convert string representation of tensor into computer object: ‘...’.$^t$. Here is a usage example:

```
def tensor = 'F^{A}_{B \mu \nu}*a'.t
```

The general conventions on the expression input are the same as in many other CASs. The main difference arises from the tensorial nature of Redberry. The following code gives an idea of the valid Redberry syntax for tensors:

```
def t
2 //same, but without braces
```

\[^1\text{In the Java API this conversion is performed using static method } \text{Tensors.parse(String)}.\]
As one can see, in many cases curly braces can be omitted when inputting the indices of tensor. Braces are necessary when one needs to separate indices by spaces, or input a subscripted index. Arbitrary types of indices are not allowed in the current version of Redberry: there are 8 pre-defined types including Latin and Greek indices (both lower and upper cases) and indices with strokes. For further information about indices see Sec. 2.5. In contrast, there is no restrictions on tensor names, so we can input a tensor with Greek name written in the L\LaTeX notation, like in line 9 or 11 (both lines will produce the same tensor).

The notation for functions (both user defined and built-in scalar functions: sin, \cos etc.) is the same as in Wolfram Mathematica:

//user defined tensor field
t = 'F^a_bcd[G^a_bc, p^a]'.t
//some Redberry built-in functions
t = 'Sin[m**2 - p_m*p^m] - Log[x/2]'.t
t = 'Power[a, b]'.t //same as 'a**b'.t

It is important to note, that according to the meaning of scalar functions their arguments must always be scalar, so the following code will exit with error:

\begin{verbatim}
def t = 'Sin[A_m]'.t
\end{verbatim}
\begin{verbatim}
    ▷ IllegalArgumentException
\end{verbatim}

All Redberry objects have a default string representations. So, in order to output some expression to the stdout one can simply use println keyword in Groovy scripts. Additionally to the default Redberry output format, there are several others. The following example illustrates their usage:\footnote{\texttt{cc.redbery.core.context.OuputFormat.*} should be added to static imports.}

\begin{verbatim}
def t = 'F_{mn}^{\alpha\beta}/(a+b)'.t
//default output format
println t
▷ (b+a)**(-1)*F_{mn}^{\alpha\beta}
//LaTeX format
println t.toString(LaTeX)
▷ \frac{1}{(a+b)} F_{mn}^{\alpha\beta}
//UTF8 format will print greek characters
println t.toString=UTF8
▷ (a+b)**(-1)*F_{mn}^{\alpha\beta}
\end{verbatim}
By default, Redberry uses its own output format, so line 3 is equivalent to

```
println t.toString(Redberry)
```

This line produces string in Redberry input format, so the result can be parsed back into Redberry. The \textit{LaTeX} format produces simple \texttt{\LaTeX} code that could be parsed by \texttt{\LaTeX}. If console supports unicode characters, then the \texttt{UTF8} format can be used to print Greek letters as is, while in other aspects this format is the same as default. Finally, the \textit{WolframMathematica} format produces the expression in form that can be directly parsed by Wolfram Mathematica. It is useful for further manipulations in Mathematica with symbolic (without indices) expressions.

### 2.2 Tensors

Redberry is written in Java and makes extensive use of its object-oriented features. Though user may not be familiar with object oriented programming (OOP) to use Redberry, it is still useful to understand primitive object types that are used in the CAS. There are three central object types in Redberry: \texttt{Tensor}, \texttt{Indices} and \texttt{Transformation}. Objects of the same class share common properties and can be manipulated in a common way\footnote{In fact, Java internals of Redberry are highly introduced into main user interface (Groovy scripting). Despite the fact that only two developers created Redberry, the usage of Groovy as a general programming environment allowed us to implement rather convenient way for flexible Redberry usage. One of the main targets of Redberry is availability to the broader audience. So, authors tried to introduce simplicity and sameness as much as possible into the internal architecture as well as into Groovy binding. When architecting Redberry the authors were guided by intention to mimic the natural sense of tensorial expressions.}. Tensors are considered in this section, while indices and transformations are discussed in Sec. 2.5 and Sec. 3 respectively.

Each mathematical expression in Redberry is a \texttt{Tensor}. Any \texttt{Tensor} have \texttt{Indices} and content (summands in case of sum, arguments in case of functions, etc.; thus, tensors in Redberry are containers of other tensors). Here is how these properties can be accessed via Groovy syntax:

```groovy
def product = 'F^{A}_{B \mu \nu}*a'.t
println product[0] 
println product.size() 
product.each{ println it }
def sum = 'A^{ijk} * B_{i} + N^{j} * M^{k}'.t
println sum.indices
```

```
The meaning of the above lines of code is pretty evident. First two examples demonstrate two ways of accessing (getting by index and enumerating) child elements of a tensor. The last example demonstrates that indices of a tensor can be taken via `indices` property.

In order to create tensors programmatically, Redberry defines all arithmetic operations for `Tensor` objects (mathematical expressions):

```python
def t = 'A_i+C_i'.t, u = 'G^i'.t, k = 'S^m_m'.t
println t*u+k
```

However, such syntax may be inconvenient when dealing with indexed objects, since names of variables do not reflect structure of indices of expressions.

Tensors are immutable and modification operations return the new instance:

```python
def t = 'A_i+C_i'.t
def x = t.set(0, 'N_j*D^j_i'.t)
println x
```

### 2.3 Einstein notation

At this point the basic conventions arising from Einstein notation should be clarified. As was previously mentioned, Redberry distinguishes covariant (lower) and contravariant (upper) indices. The corresponding property of single index we call `state`. Two indices are considered to be contracted if and only if they have similar names and types but different states. As a consequence of this convention there are some natural restrictions on general structure of the expressions.

First of all, the following notation (used, for example, in Euclidean space) is illegal in Redberry:

```python
2*Sin[-y]+2*x
```

---

1 The implementation uses Groovy operator overloading, while the Java equivalents are placed in `Tensors` class.
The correct input for the above tensor should be the following:

```python
def t = 'F_a^a'.t
```

In addition, the error occurs when expression is meaningless because of several indices with the same name and state coexist in the same product:

```python
//meaningless expression
def t = 'F_ij*M^i*N^j*K^j'.t
```

The entire architecture of Redberry built in such a way that the above illegal situations can not ever arise during manipulations and user do not need to take care about it.

Other thing arises with dummy indices in products, where one or more multipliers are sums. It is convenient to write

\[ F^{\mu\nu}(A^{\alpha\beta} + M^\mu N^{\mu\alpha\beta}) \]

There is no problem here, since dummy index \( \mu \) is in the scope of sum, which "outer"\(^1\) indices are upper \( \alpha,\beta \), while the first multiplier indices are lower \( \mu,\nu \). However, if we try to expand brackets in this expression naively (i.e. without relabeling of dummy index \( \mu \)) we shall face the ambiguity as described in the previous examples. Thus it is better to relabel such dummy indices in sums right after parsing. Such transformation is automatically performed for every input expression in Redberry. Consider the following code:

```python
def t = 'F_mn*(A^ab + M_m*N^mab)'.t

println t
```

As we can see, dummy index \( m \) was automatically renamed to \( c \), because conflict with free index of the whole tensor was detected. Similar behaviour is also valid for powers in products:

```python
def t = 'F_a*(A^a*B_a)**2'.t

println t
```

As we can see, dummy index \( a \) was automatically renamed to \( b \), because conflict with free index of the whole tensor was detected. Similar behaviour is also valid for powers in products:

2.4 Standard form of mathematical expressions

A core function of any CAS is its ability to reduce arbitrary expression to some standard form (SF), which is then used everywhere in manipulations. This approach facilitates comparison and matching of expressions and gives a way for more robust and fast algorithms of almost all transformations. Redberry uses the same paradigm, so any intermediate and resulting expression is guaranteed to be in the SF.

Consider the following examples, which give an idea of SF in Redberry:

\(^1\)If consider sum as a one tensor \( T^{\alpha\beta} = A^{\alpha\beta} + M_\mu N^{\mu\alpha\beta} \)
The first line demonstrates that Redberry performs the reduction of similar terms. The second — that same (to within a sign) multipliers are collected into powers or reduced. Numbers are always collected and reduced if possible. If the expression contains floating-point numbers, then it will be completely reduced (calculated). This behaviour is similar to the majority of symbol-oriented computer algebra systems and needs no more detailed explanation.

The additional conventions on standard form arises in expressions that contain tensors. The most remarkable convention is on the SF of sum. It is best to demonstrate it by example:

As one can see, Redberry tries to factor out parts of products which contain all multipliers which have nonzero number of "outer" indices\(^1\).

The other important detail is the ordering of summands and multipliers within sums and products. Elements of sums and products are sorted by their 32 bit hash codes. Hashes for simple tensors (e.g. \(x\) or \(k_p\)) are generated randomly at each Redberry run, while the hashes of complex tensors (e.g. \(\sin[x] \cdot k_i\)) are calculated according to certain complex rules\(^2\). Usage of the pseudorandom generator allows to obtain nearly uniform distribution of hashes of tensors, which significantly improves performance. However, drawback of this approach is that ordering of expressions changes from run to run, and e.g. product \(a \cdot b \cdot c\) will be sorted differently at different runs (\(b \cdot c \cdot a\) or \(b \cdot a \cdot c\) etc.). Of course, all similar expressions will have similar ordering in the current session. Still, Redberry have tools to fix the seed of pseudorandom generator inside, so that expressions will have equal ordering from run to run.

In contrast to many other tensors-oriented CASs, like xAct [5] or Cadabra [1], Redberry do not pay attention to the so-called indices canonicalization (sorting) problem. It uses fundamentally different approach for the problem of tensors comparison, which internally takes into account symmetry properties of tensors and does not depend on any particular canonical form of tensor indices.

---

\(^1\)Multipliers like \((\kappa^\mu_{\mu} + \sigma^{\alpha}_{\alpha})\) are considered to have no "outer" indices, wile tensors like \(x^{m}m\) are considered to have "outer" indices, because they contribute to a whole product indices.

\(^2\)The most important property of hash functions defined for complex tensors is its "insensibility" for particular names of indices but "sensibility" for their contractions. So, renaming of dummy or free indices is not affecting hash code, but changing the structure of contractions (e.g. contraction of two free indices) does.
In conclusion, authors want to emphasize that user can rely on fact that during any calculation any expression in Redberry is reduced to SF. This fairly simplifies implementation of custom transformations and algorithms.

2.5 Indices

Presence of indices of expressions is a main distinguishing property of tensor-oriented CASs. As mentioned above, there is an `indices` property defined for each expression in Redberry\(^1\). Returned object is an object of type `Indices` and have a number of methods and properties to work with. Here are some examples of possible operations with indices objects\(^2\):

```python
1. def t = '2*x_am*f^m*(a^n+b^n)'.t
2. def ind = t.indices
3. println ind
4. println ind.size() ▷ 4
5. println ind.free ▷ ^{n}_{a}
6. println ind.inverted ▷ ^{am}_{mn}
7. println ind.upper as Indices ▷ ^{mn}
```

Methods used in the above example, are inherent in any indices object. Their names clearly implies their meaning\(^3\).

Although the presence of indices is inherent in all tensors, different types of expressions have different subtypes of `Indices` objects. This difference arises from availability or unavailability of information about indices order. Consider indices of simple tensor:

```python
1. def simple = 'F_{mn}^{\beta\alpha}_{ba\alpha}'.t
2. println simple.indices
3. println _{mnba}^{\beta\alpha}_{\alpha}
```

We shall call indices of simple tensors as "simple indices". For "simple indices" the order of indices is defined, which is its main distinguishing property. In other words, permutation of indices will result in changing of mathematical sense of the particular simple tensor (unless this tensor is symmetric with respect to this permutation, see Sec. 2.6). However, as we previously mentioned,

---

1. `getIndices()` method in Java API
2. The last line in this example requires `cc.redberry.core.indices.Indices` to be added to imports.
3. Full list of indices properties and methods as well as corresponding documentation can be found in the JavaDocs for the `Indices` class (see full documentation on Redberry web site).
each index belongs to some index type (e.g. Greek upper/lower case or Latin upper/lower case, etc.). Indices of different types are considered to have different mathematical nature (e.g. Greek indices are Lorentz, Latin are SU(N) etc.), so the relative position of indices with different types is not important. Thus, Redberry sorts indices of simple tensors according to their types, preserving the relative order of indices belonging to the same type (see the above example). Indices of 

Another type of indices is inherent in all other types of tensors. Consider the following product:

```
def pr = 'F_{mn}*F^{\beta\alpha}*F_{ba\alpha}'.t
println pr.indices
```

From the mathematical point of view, order of product indices is undefined. This allows to set up some certain ordering rules (mainly for technical reasons, related to performance). As one can see from the example, all indices are sorted according to the following rules: first upper then lower, first Greek then Latin, in each group of indices with the same type indices are sorted in lexical order. The similar rules are adopted for sums:

```
def sum = 'R^a_amn^\alpha+K^i_inm^\alpha'.t
println sum.indices
```

The only difference, is that according to the sense of sum, its indices are only free indices.

All methods from the Listing 2.5 (.inverted, .free etc.), return objects with the same rules of ordering as in the initial indices object.

**Single index** At the low-level, Redberry stores each single index as 32-bit integer which encodes all information about index: state, type and name (serial number in the alphabet). However, there are many utility methods to operate with single index\(^1\), so, the user is free from knowing the concrete bits layout. In order to illustrate some possible manipulations with indices in Groovy scripting, consider the following advanced example\(^2\):

```
Listing 2 : Get contractions between two tensors
def getDummy = {
def ind1 = t1.indices.free,
    ind2 = t2.indices.free,
    dummy = []
    for (i in 0..ind1.size(indexType) - 1)
        for (j in 0..ind2.size(indexType) - 1)
            if (areContracted(ind1[indexType, i], ind2[indexType, j]))
                dummy << getNameWithType(ind1[indexType, i])
        dummy as int[]
    }\(^1\)see IndicesUtils class
\(^2\)The following classes from cc.redberry.core.indices package should be added to imports: Indices, IndexType.* (statically) and IndicesUtils.* (statically).
```

In this example we define function which takes two tensors, index type and returns an array of contracted indices, having specified type, between specified tensors. The information about available index types is placed in the `IndexType` enum. Of course, the above code is not the most efficient implementation of such method, but aims to emphasize the ease of accessing low-level Redberry infrastructure.

2.6 Symmetries of tensors

The next distinctive feature of tensors is their symmetries. Consider symmetries under permutations of indices. Permutational symmetries in Redberry can be defined for indices of simple tensors and tensor fields. For example, in the following code, the symmetries are defined for Riemann tensor and then all possible permutations of indices (which follows from those specified) are enumerated:

```python
def t = 'R_{abcd}'.t
addSymmetry(t, 2, 3, 0, 1)
addAntiSymmetry(t, 1, 0, 2, 3)

//printing all possible permutations for this tensor
for (s in t.indices.symmetries)
    println s
```

As one can see, there are eight permutational symmetries (including identity), which can be obtained by combining the basic symmetries. Redberry supports permutational symmetries and anti-symmetries. Tensor with only one type of indices was used here for simplicity, but in case of several types of indices, symmetries can be set for each index type separately\(^1\). The example also shows method to access basis symmetries.

Once set, symmetries of tensor affects all further manipulations with it. The following example is a good demonstration of this feature (code from the previous example assumed to be already executed):

\(^1\)see JavaDocs for details
Here zero was returned right after parsing. This is because Redberry automatically reduces sums to the standard form and
\[ R^{abcd} R_{efdc} R^{ef}_{ab} = - R_{rc}^{df} R_{ab}^{rc} R_{fd}^{ba} \]
according to the specified symmetries. Such architecture requires user to set all symmetries of simple tensor before it will be parsed inside any complex structure like sum or product. If this criteria is not fulfilled, results of calculations may be erroneous.

Another restriction concerns sequential addition of symmetries. Consider the following code:

```python
def t = 'R_abcd'.t
addSymmetry(t, 2, 3, 0, 1)
addAntiSymmetry(t, 1, 0, 2, 3)
addAntiSymmetry(t, 3, 2, 1, 0)
```

Exception is thrown on the last line. It is easy to verify, that permutation used in this line is a combination of two previous, but sign is different. Thus, some combinations of symmetries and anti-symmetries can come into conflict, which causes an error in Redberry.

Redberry provides tools to find permutational symmetries of complex tensors. Consider the following example:

```python
addAntiSymmetry('R_abc', 1, 0, 2)
addSymmetry('A_ab', 1, 0)
def t = '(R_abc*A_de + R_bde*A_ac)*A^ce + R_adb'.t
def symmetries = findIndicesSymmetries('_abd'.si, t)
for (s in symmetries)
    println s
```

The first permutation is identity, while the second means that if \( R_{abc} = - R_{bac} \) and \( A_{ab} = A_{ba} \), then tensor
\[ T_{abd} = (R_{abc} A_{de} + R_{bde} A_{ac}) A^{ce} + R_{adb} \]
satisfies the symmetry property \( T_{abd} = - T_{dab} \).

While the permutational symmetries of tensors are well covered in a number of tensors-oriented CASs, the so-called multi-terms symmetries like Bianchi identities, are faintly covered or absent at all in the majority of existing systems. In fact, authors know only one system — Cadabra [1], which fully supports multi-term symmetries. The basic idea utilized by Cadabra system is usage

1. `cc.redberry.core.utils.TensorUtils.*` should be added to static imports.
2. Static method `findIndicesSymmetries(...)` is defined in class `TensorUtils`. This method takes a simple indices as the first argument in order to specify the relative order of indices, for which one want to find symmetries.
of Young tableau projectors to reduce expressions to the simplified form. Consider the following identity:

\[ W_{uvsw}W_{tvqw}W_{ptru}W_{pqrs} - W_{svuw}W_{rvtw}W_{pqtu}W_{pqrs} = W_{spda}W_{mscd}W_{npbc}W_{mnab} - \frac{1}{4}W_{nsdc}W_{mpcd}W_{psba}W_{mnab} \]  

(1)

where \( W_{abcd} \) is a Weyl tensor. In order to prove this identity using a Young projector, one needs to apply the following substitution to the above expression:

\[ W_{abcd} = \frac{1}{3} (2W_{abcd} - W_{adbc} + W_{acbd}) \]  

(2)

This identity is derived from the Ricci cyclic identity and ordinary permutational symmetries of the Weyl tensor. The following Redberry code proves the identity (1):

**Listing 3 : Multi-term symmetries**

```python
addAntiSymmetry('W_abcd', 0, 1, 3, 2)
addSymmetry('W_abcd', 2, 3, 0, 1)
def t = ('W^n^p^q^r_s*W_p^t^u_r^w*W_tv^qw*W_u^vs_w' + ' - W^n^p^q^r_s*W_p^qtu_r^w*W_rvtw*W^sv_u^w' + ' - W_mn^ab*W^n_pb^c*W^ms_cd*W_s^pd_a' + ' + 1/4*W_mn^ab*W^ps_ba*W^m_p^c_d*W^n_s^d_c').t
def s = 'W_mnpq = 1/3*(2*W_mnpq - W_mqnp + W_mpnq)'.t
def r = (s & Expand) >> t
println r
```

At the moment Redberry have no built-in functionality to construct substitutions like (2) based on the Young projectors. But, as could be seen from the above example, if one defines the corresponding substitution manually, then Redberry allows to work with multi-term symmetries in the Cadabra way. Authors plan to incorporate support of Young projectors in the next release of Redberry.

### 3 Transformations

#### 3.1 Applying and manipulating transformations

All transformations in Redberry are first-class objects (which means that they can be assigned to variables) and share a common way for their applying and manipulating. Consider the syntax for applying transformation to mathematical expression:

```python
def tr = Expand
def t = '(A_k+B_k)*c'.t
def rr = tr >> t,
    rl = t << tr
assert rr == rl
println rr
```

---

1. This example is taken from Section 2.2 of the 'Cadabra: reference guide and tutorial' by Kasper Peeters, which is available on Cadabra web site.
As it seen from the example, transformations are applied using left shift `<<` or right shift `>>` operator. It should be noted that both operators are left-associative, which means that, for example, \( A << B << C \) will be treated as "apply transformation B to tensor A, and then apply C to the resulting tensor", while \( C >> B >> A \) will be treated as "apply C to B and then apply resulting transformation to tensor A". The latter behaviour is not what one can expect, so it is better to use a special & operator, which allows to join a set of transformations into a single one:

\[
\begin{align*}
& \text{def } t = '\( (a+b) \times c \)' . t \\
& \text{def } expandAndSubs = \text{Expand} & \text{ 'c = a + b' . t} \\
& \text{println } expandAndSubs >> t \\
& \quad \triangleright a \times (a+b) + b \times (a+b)
\end{align*}
\]

\[
\begin{align*}
& \text{def } subsAndExpand= 'c = a + b' . t & \text{Expand} \\
& \text{println } subsAndExpand >> t \\
& \quad \triangleright a**2 + 2*a*b + b**2
\end{align*}
\]

Transformations (like tensors) are immutable in Redberry. Some transformations may take required or optional arguments using square brackets:

\[
\begin{align*}
& \text{def } eliminateWhileExpand = \text{Expand}[\text{EliminateMetrics}], \\
& \quad x = '\( g_{mn} + d_m^a g_{na} \) \times f^{mn}' . t \\
& \text{println } eliminateWhileExpand >> x \\
& \quad \triangleright 2*_{m}^{f_m}
\end{align*}
\]

\[
\begin{align*}
& \text{def } diff = \text{Differentiate}['x_m'], \\
& \quad y = '\_m*x^m' . t \\
& \text{println } diff >> y \\
& \quad \triangleright 2*x^m
\end{align*}
\]

In this example, the Expand transformation takes an optional parameter, which is a transformation to be applied on each level of expand procedure. In contrast, the argument of Differentiate transformation is required. In both cases a new object will be created and assigned to a corresponding variable. The meaning of arguments is specific for each particular transformation and will be discussed in further sections.

### 3.2 Selected general-purpose transformations

#### 3.2.1 Substitutions

The most frequent transformation in all computations is a substitution. Here we shall discuss the usage aspects of substitutions, while the idea of the underlying algorithms can be found in Sec. 5.1.

---

1Transformations covered in this section are part of [redberry-core](https://github.com/redberry/redberry-core), their Java implementation can be found in [cc.redberry.core.transformations](https://github.com/redberry/cc.redberry.core.transformations) package.
The very important feature of any tensors-oriented CAS is automatic relabelling of dummy indices in the case of dummy indices clash. Redberry takes care about it in all types of substitutions. Consider, for example, the following simple substitution:

\[ x = x_a^a \quad \text{in} \quad (x f_a + y_a)(x f_b + z_b) \]

Here is a code to perform this substitution in Redberry:

```
def s = 'x = x_a^a'.t
def t = '(x*f_a + y_a)*(x*f_b + z_b)' .t
println s >> t
```

\[ (x_{d}^d*f_{a}+y_{a})*(x_{c}^c*f_{b}+z_{b}) \]

As one can see, the appropriate relabelling was performed automatically.

Redberry supports substitutions of tensor fields and automatically performs matching of the arguments of functions during substitution:

```
def s = 'F_{ij}[x_{m}, y_{m}] = x_{i}*y_{j}'.t
def t = 'T^{ab}F_{ab}[p_{a}-q_{a}, p_{a}+q_{a}]'.t
println s >> t
```

\[ T^{ab}*(p_{a}-q_{a})*(p_{b}+q_{b}) \]

If tensor field depends on indexed argument, then some ambiguities arises when mapping the arguments. For example, the following substitution:

\[ F_{i}(x_{mn}) = x_{ik}f^{k} \rightarrow F_{k}(x_{i}y_{j}) \]

can be performed in two different ways: (a) matching \( x_{ij} \rightarrow x_{i}y_{j} \) gives \( x_{k}y_{j}f^{j} \), (b) matching \( x_{ij} \rightarrow x_{j}y_{i} \) gives \( y_{k}x_{j}f^{j} \). This is because the indices of product \( (x_{i}y_{j}) \) are not ordered by their nature. To explicitly specify the matching rule in such ambiguous situations, Redberry allows to enter the correspondence of indices in the field arguments:

```
def s = 'F_{i}[x_{mn}] = x_{ik}*f^{k}'.t
def t = 'F_{k}[x_{i}*y_{j}:_ji]'.t
println s >> t
```

\[ x_{a}*y_{k}*f^{a} \]

The second line in this example is actually equivalent to

```
def t = 'F_{k}[x_{i}*y_{j}:_ij]'.t
```

because indices of product are sorted according to the rules discussed in Section 2.5.

As well as simple tensor and tensor field substitutions, Redberry fully supports all other types of substitutions, taking into account both indices symmetries and indices contractions. Consider the following complex example. Let’s apply the following substitution

\[ f_{m} + R_{bma} F^{ba} - R_{ijm} F^{ij} = R_{bam} F^{ab} \]
to tensor

\[ f_i + R_{ijk}F^{jk} + R_{ijk}F^{kj} - R_{kij}F^{jk}, \]

where \( R_{abc} \) is antisymmetric: \( R_{abc} = -R_{cba} \). It is clear that the sum over first, third and fourth terms can be rewritten in the following way:

\[ f_i - R_{kij}F^{jk} + R_{ijk}F^{kj} = f_i + R_{ika}F^{ka} - R_{kij}F^{jk}, \]

where the last step is a simple dummy relabelling. After this steps, it becomes obvious that these terms match the l.h.s. of the substitution. In Redberry one can e.g. do

```python
addAntiSymmetry('R_mnp', 2, 1, 0)
def s = 'f_m + R_bma*F^ba - R_ljm*F^lj = R_bam*F^ab'.t
def t = 'f_i + R_ijk*F^jk + R_ijk*F^kj - R_kij*F^jk'.t
println s >> t
.isDirectory
```

We see, that Redberry matched the l.h.s. of the substitution in tensor \( t \) and automatically reduced the resulting sum to the standard form, which, in turn, gives zero.

Redberry takes into account not only predefined symmetries of simple tensors, but also symmetries of any complex expression, which arise from its structure. For example:

```python
def s = 'K_a*(A^ab - A^ba) = F^a*A_a^b'.t
def t = 'K_p*(A^qp - A^pq) + F^b*A_b^q'.t
println s >> t
```

The result is zero since tensor \( (A_{cb} - A_{bc}) \) is antisymmetric. As well, Redberry takes care about symmetries of indexless objects:

```python
def c = 'Cos[a - b] = c'.t,
s = 'Sin[a - b] = s'.t,
t = 'x = Cos[b - a]**3 + Sin[b - a]**3'.t
println (c | s) >> t
```

As one can see, the built-in definitions of sin and cos are set up to be odd and even respectively.

There is an important note on applying several substitutions at a time using the joining of transformations. Substitutions joined with \& operator will be applied sequentially. However, sometimes it is necessary to apply several substitution rules "simultaneously". Consider the following example:

```python
def X2YandY2X = 'x=y'.t & 'y=x'.t,
X2YorY2X = 'x=y'.t | 'y=x'.t,
tensor = 'x+2*y'.t
println X2YandY2X >> tensor
```

```plaintext
3*x
```

```python
println X2YorY2X >> tensor
```

```plaintext
18
```
\[ y + 2 \times x \]

The first transformation (X2YandY2X) means just sequential applying of two provided substitutions, while the second (X2YcrY2X) performs both substitutions "simultaneously".

The last thing need to be discussed in this section is that the expression like

```python
def t = 'x = a + b'.t
```

is a mathematical expression (Tensor) and transformation (Transformation) at the same time. In order to construct such expression programmatically from a given l.h.s. and r.h.s. one can use the following syntax:

```python
def lhs = 'x'.t, rhs = 'a + b'.t,
subs = lhs.eq rhs
println subs >> 'x**2'.t
```

\[ (a + b) \times 2 \]

### 3.2.2 Differentiate

This transformation allows to take derivatives with respect to indexed objects. Consider the examples:

```python
def tensor = 'Sin[f_ab*f^ab]'.t
println Differentiate['f_mn'] >> tensor
```

\[ 2 \times \text{Cos}[f^{ab} \times f_{ab}] \times f^{mn} \]

//derivative of antisymmetric tensor
setAntiSymmetric('R_{ab}')
println Differentiate['R_{ab}'] >> 'R_{mn}'.t

\[ \frac{1}{2} \times (d_{m}^{a} \times d_{n}^{b} - d_{n}^{a} \times d_{m}^{b}) \]

The Differentiate transformation takes care about dummies relabelling and symmetries of tensors. The following convention is adopted:

\[
\frac{\delta T_{m_1...m_k}}{\delta T_{n_1...n_k}} = \frac{1}{N} (\delta_{m_1}^{m_1} \times \delta_{m_k}^{m_k} \text{ + permutations}),
\]

where \(N\) is a number of elements in the sum, and r.h.s of this expression have the same symmetries as l.h.s.

It is also possible to pass additional transformations which should be applied after each step of differentiation (mainly for performance reasons):

```python
//setting up symmetries of Riemann tensor
addAntiSymmetry('R_{abcd}', 1, 0, 2, 3)
addSymmetry('R_{abcd}', 2, 3, 0, 1)
def tensor = 'R^{acbd}*Sin[R_{abcd}*R^{abcd}]'.t,
varl = 'R^{ma}_{m}b'.t,
```

19
var2 = 'R^mc_m^d'.t

def diff1, diff2
timing {
  //take second derivative and then simplify
  diff1 = (Differentiate[var2, var1] & ExpandAndEliminate) >> tensor
  ▶ Time: 1338 ms.
}
timing {
  //take second derivative and simplify permanently
  diff2 = Differentiate[var2, var1, ExpandAndEliminate] >> tensor
  ▶ Time: 14 ms.
}
assert diff1 == diff2

As one can see from this example, if ExpandAndEliminate (which simply expand out brackets and eliminates contractions with metrics and Kronecker deltas) performs on the each level of the differentiation, the derivative will be taken much faster, then if it applies only to the result.

3.2.3 EliminateMetrics

This transformation eliminates metric tensors and Kronecker deltas, which are contracted with other tensors:

def tensor = 'g_nm*A^m*d^n_a'.t
println EliminateMetrics >> tensor
▶ A_a
tensor = ('g^mn*g^ab*g^gd*(p_g*g_ba + p_a*g_bg)'
  '*(p_m*g_dn + p_n*g_dm)').t
//eliminate metrics in D dimensions
println ((EliminateMetrics & 'd_a^a = D').t) >> tensor
▶ 2*(1+D)*p^{d}*p_{d}

3.2.4 Expand

There are several transformations which expand out products and integer powers of sums: Expand, ExpandAll, ExpandNumerator and ExpandDenominator. These well-known transformations needs no special explanation except the fact that they can take an additional transformations as an arguments to be applied on the each level of expand procedure. Consider the following example:

def tensor = ('(g_af*g_bc+g_bf*g_ac+g_cf*g_ba)*(T_d*T_e+g_de)' +
  '*(g^db*g^ae + g^de*g^ab)').t
//eliminates metrics in four dimensions
def eliminate = EliminateMetrics & 'd_a^a = 4'.t
//expand and then eliminate
def r1 = (Expand & eliminate) >> tensor
println r1
▷ 2*T_{c}*T_{f}+30*g_{fc}+7*g_{fc}*T_{e}*T^{e}

//eliminate while expand
def r2 = Expand[eliminate] >> tensor
assert r1 == r2

As we see, lines 6 and 8 produce the same result, but the latter spends less time in the calculation because it applies additional simplifications not only to the final result, but to all intermediate tensors, drastically reducing their complexity.

As was many times noted, all operations in Redberry are safe from conflicts of dummies, and, for example, Expand transformation automatically renames dummy indices of integer powers in the resulting sum:

//dummy indices in power
def t = ‘(a+b)/(A_m^m+B)**2*A_a’ .t
println ExpandDenominator >> t
▷ (a+b)*(2*B*A_{b}^{b}+A_{m}^{m}*A_{b}^{b}+B**2)**(-1)*A_{a}

3.2.5 Factor

This transformation factors symbolic polynomials over the integers in expressions\(^{1}\). Consider the examples:

//univariate polynomial
def t = ‘1 + 2*x + x**2’ .t
println Factor >> t
▷ (1+x)**2

//multivariate polynomial
t = ‘2*x**3*y - 2*a**2*x*y - 3*a**2*x**2 + 3*a**4’ .t
println Factor >> t
▷ (x+a)*(x-a)*(-3*a**2+2*y*x)

//tensorial expression with symbolic polynomial parts
t = Expand >> ‘(a+b)**4*F_mn + (x**6-y**6)*R_mn’ .t
println Factor >> t
▷ (a+b)**4*F_mn+(x+y)*(x-y)*(x*y+x**2+y**2)*(-x*y+x**2+y**2)*R_mn

Factor traverses the expression from head to children and factors all symbolic (indexless) polynomials (both univariate and multivariate) and rational functions. So, in the case of completely symbolic expression it applies only to the top algebraic level. In the case of rational polynomials factorization is taken from free and open source Java Algebra System [6, 7] designed by Heinz Kredel.

\(^{1}\)The code of polynomials factorization is taken from free and open source Java Algebra System [6, 7] designed by Heinz Kredel.
expression, Factor first calls Together (Sec. 3.2.6), then factors numerator and denominator.

It should be noted, that in the current version of Redberry, performance of Factor transformation in case of multivariate polynomials may be low in some cases and some huge expressions may not be factored completely. Also Factor applies only to expressions, which does not contain trigonometric functions, tensor fields or non integer powers of variables. These issues will be fixed in the upcoming release of Redberry.

3.2.6 Together

There are two transformations, which put terms in a sum over a common denominator: Together and TogetherFactor. The last one also cancels symbolic (without any indices) factors in the result. Consider the examples:

```plaintext
1 def t = 'a/b + c/d'.t
2 println Together >> t
   ▶ b**(-1)*d**(-1)*(b*c+a*d)
3 t = 'x**2/(x**2 - 1) + x/(x**2 - 1)'.t
4 println Together >> t
   ▶ (-1+x**2)**(-1)*(x+x**2)
5 println TogetherFactor >> t
   ▶ (1+x)**(-1)*(-1+x)**(-1)*(x+x**2)
7 t = 'f_m/a + k_m/(f_m*f^m)'.t
8 println Together >> t
   ▶ a**(-1)*(f_(a)*f^(a))**(-1)*(f_(b)*f^(b)*f_(m)+a*k_(m))
```

As it seen from the last example, Together effectively relabel dummy indices when it is necessary.

3.3 Selected physical transformations

3.3.1 DiracTrace

This transformation calculates the Dirac trace of expression in four dimensions. First, consider traces without $\gamma_5$:

```plaintext
1 //set up matrices
defineMatrix 'G_a', Matrix1.matrix
2 //dirac trace transformation
3 def dTrace = DiracTrace['G_a']
```

1Transformations covered in this section are part of redberry-physics. The Java implementation can be found in cc.redberry.physics.feyncalc package.

2cc.redberry.core.indices.IndexType.* should be added to static imports
//trace of product of matrices
def t = 'Tr[G_a*G_b]'.t
println t
▷ G_a^a'_b'*G_b^b'_a'

//calculate trace
println dTrace >> t
▷ 4*g_ab

t = 'Tr[(p_a*G^a + m)*G_m*(q_a*G^a-m)*G_n]'.t
println dTrace >> t
▷ 4*p_{m}*q_{n}+4*p_{n}*q_{m}-4*m**2*g_{mn}-4*p^{a}*g_{mn}*q_{a}

t = 'Tr[G_a*G_b*G_m*G^a*G_n*G_f*G^b*G^f]'.t
println dTrace >> t
▷ -32*g_mn

The first line tells Redberry to consider tensor \( G_a \) as matrix with additional one upper and one lower index of type Matrix1 (Latin lower letters with strokes). This means that everywhere at the input Redberry will consider \( G_a \) as matrix \( G_a^{i'_j'} \) and use matrix multiplication rules for the products of such tensors (see Appx. A for details). Line 4 defines \texttt{DiracTrace} transformation with specified notation for gamma matrix.

Consider traces with \( \gamma_5 \):

---

//set up matrices
defineMatrices 'G_a', 'G5', Matrix1.matrix
//setting up symmetries of Levi-Civita
setAntiSymmetric('e_abcd')
//dirac trace transformation
def dTrace = DiracTrace['G_a', 'G5', 'e_abcd']

def t = 'Tr[G_a*G_b*G_c*G_d*G5]'.t
println dTrace >> t
▷ -4*I*e_{abc}d

t = 'Tr[G_a*G_b*G_c*G^d*G_e*G_f*G5]'.t
println dTrace >> t
▷ 4*I*g_ab*e_{fc}d_e+4*I*e_{d_c}d_e+fabe+4*I*g_fb*e_{c}d_ea-4*I*g_ce*e_{fab}d+4*I*e_{d_e}fabc-4*I*g_fa*e_{c}d_{eb}

t = 'Tr[(p_a*G^a + m)*G_m*G5*(q_a*G^a-m)*G_n]'.t
println dTrace >> t
▷ -4*I*p_{b}*q_{n}+4*I*e_{a}d_{n}+4*I*e_{b}d_{m}
---

As we see from line 6, when expression contains \( \gamma_5 \), it is also necessary to specify the notation of \( \gamma_5 \) (\( G5 \)) and Levi-Civita tensor (\( e_{abcd} \)) to \texttt{DiracTrace}.

### 3.3.2 UnitaryTrace

This transformation calculates traces of unitary matrices:
//set up matrices
defineMatrix 'T_A', Matrix2.matrix

//structure constants are antisymmetric
setAntiSymmetric('f_ABC')

//d-constants are symmetric
setSymmetric('d_ABC')

//unitary trace transformation
def uTrace = UnitaryTrace['T_A', 'f_ABC', 'd_ABC', 'N']

//trace of product of matrices
def t = 'Tr[T_A*T_B].t
println t
   ▶ T_A^A'_B'*T_B^B'_A'

//calculate trace
println uTrace >> t
   ▶ (1/2)*g_AB

//simplify in Euclidean space in three dimensions
println LeviCivitaSimplify.euclidean['e_abc'] >> t
   ▶ 2*d^d_c

As one can see, it is necessary to specify the notation used for structure and d- constants (f_{ABCD} and d_{ABCD}) and dimension of unitary group (N).

3.3.3 LeviCivitaSimplify
This transformation simplifies combinations of Levi-Civita tensors:

//three dimensions
setAntiSymmetric('e_abc')
def t = 'e_abc*eabd'.t

//simplify in Euclidean space in three dimensions
println LeviCivitaSimplify.euclidean['e_abc'] >> t
   ▶ 2*d^d_c

//simplify in Minkowski space in three dimensions
println LeviCivitaSimplify.minkowski['e_abc'] >> t
   ▶ 2*d^d_c
\[ t = 'e_{abc} e^{abc}'.t \]

//simplify in Minkowski space in three dimensions
println LeviCivitaSimplify.minkowski['e_{abc}'] >> t
▷ 6

//four dimensions
setAntiSymmetric('e_{abcd}')
\[ t = '4*I e^h_d^fb e_{abch} e_{e^d_gf}'.t \]

//simplify in Euclidean space in four dimensions
println LeviCivitaSimplify.euclidean['e_{abcd}'] >> t
▷ 16*I e_{eagc}

\[ t = '4*I e^h_d^fb e_{abch} e_{e^d_gf}'.t \]

//simplify in Minkowski space in four dimensions
println LeviCivitaSimplify.minkowski['e_{abcd}'] >> t
▷ -16*I e_{eagc}

As one can see, it is necessary to specify whether space is considered to be Euclidean or Minkowski and the notation for Levi-Civita tensor. The difference between Euclidean and Minkowski Levi-Civita tensors appears for even number of space-time dimensions (compare lines 16 and 19). Also, as it is seen for example from line 11, the transformation automatically substitute the dimension of space, which is considered to be equal to the number of Levi-Civita indices.

4 Selected physical applications

4.1 Feynman diagrams

One of the most prominent applications of Redberry is calculation of Feynman diagrams in quantum field theory. As previously mentioned, Redberry provides several common physical transformations such as traces of Dirac gammas and SU(N) matrices, simplification of Levi-Civita combinations etc. It also provides a powerful tools for simple inputting of noncommutative matrix expressions. In this section as an example we shall consider the calculation of the well-known differential cross section of the Compton scattering in quantum electrodynamics (i.e. spinor electrodynamics, in contrast to the scalar electrodynamics, which was illustrated in Sec. 1.2). The physical background can be found in Sec. 5.5 of [8].

First of all it is necessary to note, that Redberry does not support noncommutative products of indexless variables. However, many noncommutative objects, which occur in physical calculations (e.g. spinors or Gamma matrices) have a matrix origin, which means, that they have additional matrix indices, which are usually omitted for convenience. For example, when we write product of Dirac bispinors and gammas, like e.g.

\[ T_{\mu\nu} = \bar{u} \gamma_\mu \gamma_\nu \]  

we mean, that the above quantities have additional matrix indices, which can be written explicitly:

\[ T_{\mu\nu}^{a'b'} = \bar{u}^{a'} \gamma_{\mu}^{a'} \gamma_{\nu}^{b'} \gamma_{\nu}^{c'} y'. \]  

In this expression \(a', b', c'\) are special matrix indices. From such point of view these "noncommutative" products can be represented as ordinary products of indexed objects. Such matrix indices should have a nonmetric type, which implies that it is impossible to perform raising or lowering or define a symmetry which mixes indices with different states. Redberry provides an internal facilities allowing to input matrix expressions in a convenient form like (3). The matrix
indices will be automatically inserted at parsing and subsequent processing could be made as with usual indexed expressions. Detailed description of this feature can be found in Appendix A.

So, let us turn to the physical aspects of the problem. The following lines give squared matrix element of the Compton scattering in quantum electrodynamics:\(^1\):

```
Listing 4 : Compton scattering in QED

1  defineMatrices 'G_a', 'V_i', 'D[x_m]', Matrix1.matrix,
2     'vu[p_a]', Matrix1.vector,
3     'cu[p_a]', Matrix1.covector

4  //photon-electron vertex
5  def V = 'V_m = -I*e*G_m'.t,
6  //electron propagator
7  D = 'D[p_m] = -I*(m + p_m*G^m)/(m**2 - p_m*p^m)' . t,
8  //diagram a)
9  Ma = 'cu[p2_m]*V_m*e^m[k2_m]*D[k1_m+p1_m]*V_n*e^n[k1_m]*vu[p1_m]' . t,
10  //diagram b)
11  Mb = 'cu[p2_m]*V_m*e^m[k1_m]*D[p1_m-k2_m]*V_n*e^n[k2_m]*vu[p1_m]' . t,
12  //matrix element
13  M = Ma + Mb
14  M = (V & D) >> M
15
16  def mandelstam = setMandelstam(
17      ['p1_m': 'm', 'k1_m': '0', 'p2_m': 'm', 'k2_m': '0'])
18  M = (ExpandAll & mandelstam) >> M
19
20  //complex conjugation
21  def MC = M
22  MC = 'vu[p1_m]*cu[p2_m] = vu[p2_m]*cu[p1_m]' . t >> MC
23  MC = (Conjugate & InverseOrderOfMatrices[Matrix1]) >> MC
24
25  //squared matrix element
26  def M2 = ExpandAll >> (M * MC / 4)
27
28  //photon polarizations
29  M2 = 'e_m[k1_a]*e_n[k1_a] = -g_mn'.t >> M2
30  M2 = 'e_m[k2_a]*e_n[k2_a] = -g_mn'.t >> M2
31
32  //electron polarizations
33  M2 = 'vu[p2_m]*cu[p2_m] = m + p2^m*G_m'.t >> M2
34  M2 = 'vu[p1_m]*cu[p1_m] = m + p1^m*G_m'.t >> M2
35
36  //applying trace of gamma matrices
37  M2 = DiracTrace['G_a'] >> M2
38
39  //final simplifications
40  M2 = (ExpandAndEliminate & 'd^m_m = 4'.t & mandelstam) >> M2
41  M2 = 'u = 2*m**2 - s - t'.t >> M2
42  M2 = Factor >> M2
43
44  println M2
```

\(^1\) cc.redberry.core.indices.IndexType.* should be added to static imports
The above code reproduces the standard steps of Feynman diagrams calculation and prints squared matrix element of the Compton scattering averaged over polarizations of initial particles and summed over polarizations of final particles:

\[
\frac{1}{4} \sum_{\text{spins}} |\mathcal{M}|^2 = \frac{2e^4}{(m^2 - s)^2(-m^2 + s + t)^2} \times \\
\times (-8s^2m^2t + 4s^3t + 2s^4 + t^3s + 2m^8 + 4m^4st - m^2t^3 \\
-2m^2t^2s + 3m^4t^2 - 8s^3m^2 + 12s^2m^4 + 3s^2t^2 - 8m^6s)
\]

Let’s consider the code in more detail. Firstly, in lines 1—3 we tell Redberry to consider some input objects as (noncommutative) matrices, vectors or covectors. These objects are: gamma matrices (\(G_{\alpha}\)), photon-electron vertex (\(V_i\)), electron propagator (\(D[p_m]\)), electron wave function \(u(p)\) (\(vu[p_m]\)) and its conjugation \(\bar{u}(p)\) (\(cu[p_m]\)). To make this, 
\[
defineMatrices(\ldots)
\]
defines the type of matrix indices and a type of matrix (e.g. \(Matrix1.matrix\), \(Matrix1.covector\)). Here only one matrix index type is used. As result, for example, the combination \(\bar{u}(p_1)\gamma_\mu u(p_2)\) will be treated as scalar with respect to matrix indices:

```python
def t = 'cu[p1_m]*G_m*vu[p2_m]'.t
println t
▷ cu_{a'}[p1_m]*G_{m}[a'_m]^a_{b'}*vu^b_{[p2_m]}
println t.indices.free
▷ \_\{m\}
```

while the combination \(u(p_1)\bar{u}(p_2)\) will be treated as matrix:

```python
def t = 'vu[p1_m]*cu[p2_m]'.t
println t
▷ vu_{a'}[p1_m]*cu^b'_{[p2_m]}
println t.indices.free
▷ ^b'_{[a']}
```

The next step is to build the matrix element. There are two Feynman diagrams corresponding to the Compton scattering in the leading order:
This done in the lines 5 - 14, where $e_n[k_m]$ denotes photon polarization. Important point can be seen in the line number 8, where the propagator is introduced. The identity matrix (Kronecker delta) is inserted automatically in the numerator, where the gamma matrix is added to the scalar mass:

\[
\text{println } 'm + p_i \ast G^i'.t
\]

\[
\triangleright m \ast d^{a'}_{b'} + p_i \ast G^{i a'}_{b'}
\]

At the next step (lines 17 - 19) we define mandelstam variables and expand out the matrix element. Let us consider the function `setMandelstam(...)`:

\[
\text{setMandelstam('}p1' : 'm1', 'p2' : 'm2', 'p3' : 'm3', 'p4' : 'm4'\')
\]

It takes a map 'momentum — mass of particle' as an argument and returns a collection of substitutions, followed from the rules:

\[
\begin{align*}
\text{s} &= (p_1 + p_2)^2 = (p_3 + p_4)^2 \\
\text{t} &= (p_1 - p_3)^2 = (p_2 - p_4)^2 \\
\text{u} &= (p_1 - p_4)^2 = (p_2 - p_3)^2
\end{align*}
\]

So, for example, the line 17 will produce the following substitutions:

\[
\text{println mandelstam}
\]

\[
\triangleright p_1(m) \ast p_1(m) = m^2 \\
\triangleright k_1(m) \ast k_1(m) = 0 \\
\triangleright p_2(m) \ast p_2(m) = m^2 \\
\triangleright k_2(m) \ast k_2(m) = 0 \\
\triangleright 2 \ast k_1(m) \ast p_1(m) = s-m^2 \\
\triangleright 2 \ast k_2(m) \ast p_2(m) = s-m^2 \\
\triangleright -2 \ast p_2(m) \ast p_1(m) = -2 \ast m^2 + t \\
\triangleright -2 \ast k_1(m) \ast k_2(m) = t \\
\triangleright -2 \ast k_2(m) \ast p_1(m) = u-m^2 \\
\triangleright -2 \ast k_1(m) \ast p_2(m) = u-m^2
\]

which then will be applied in further calculations.

The next step (lines 22 - 24) is to make up a complex conjugation of the matrix element. As it is known from physics, this can be done using the following relation:

\[
\left( \bar{u}(p_2) \hat{M} u(p_1) \right)^* = \bar{u}(p_1) \hat{M} u(p_2),
\]
where
\[
\hat{M} = \sum_i c_i \hat{a}_1 \hat{a}_2 \ldots \hat{a}_i
\]
\[
\hat{\tilde{M}} = \sum_i \tilde{c}_i \hat{a}_i \hat{a}_{i-1} \ldots \hat{a}_1
\]
and usual notation \( \hat{a} = \gamma_\mu a^\mu \) is used. This transformation is equivalent to two substitutions:
\[
u(p_1) \bar{u}(p_2) \rightarrow \nu(p_2) \bar{u}(p_1)
\]
\[
\hat{M} \rightarrow \hat{\tilde{M}}
\]
The first substitution is applied in the line number 23 and the second in the line number 24. The meaning of InverseOrderOfMatrices[Matrix1] transformation is clear from its name:

```plaintext
defineMatrix 'A_{\mu}', Matrix1.matrix

def p = 'A_\mu*A_\nu'.t
println p
  ▷ A_{\mu}^{a'}_{b'}*A_{\nu}^{b'}_{c'}
println InverseOrderOfMatrices[Matrix1] >> p
  ▷ A_{\nu}^{a'}_{b'}*A_{\mu}^{b'}_{c'}
```

At this point we can define squared matrix element in the line number 27 and perform the summation over photons (lines 30 - 31) and electrons (lines 34 - 35) polarizations. On the last step all products of gamma matrices are automatically converted to combinations of their traces. So the next step is to apply DiracTrace transformation. After this, it is necessary to eliminate metric tensors and apply mandelstam substitutions (lines 41 - 42). This step will produce a symbolic expression, which then can be finally simplified using the Factor transformation. The whole calculation takes less than 0.5 seconds on the hot Java Virtual Machine (on Intel Core i5 @ 2.27 Ghz).

### 4.2 One-loop counterterms of arbitrary Lagrangians

Another remarkable application of Redberry is calculation of the divergent part of the one-loop effective action of arbitrary Lagrangians. The theoretical formalism based on the extended t’Hooft and Veltman method of the background calculations was developed in [9] and successfully applied for a number of theories using REDUCE computer algebra system [10]. This algorithm calculates the one-loop counterterms for an arbitrary theory and background in four dimensions in curved space-time in the dimensional and higher derivative [11] regularizations. Redberry implements the algorithms developed in [9] for second and fourth order differential operators and provides a simple and convenient user interface\(^1\).

Let’s start with a minimal physical background. It is well-known, that one-loop effective action for a general field theory with a given action \( S[\phi] \) can be expressed in terms of derivative of action with respect to the fields:

\[
\Gamma^{(1)} = i\hbar \text{Tr} \left( \ln \frac{\delta^2 S}{\delta \phi \delta \phi_j} \right),
\]

\(^1\)The implementation is placed in cc.redberry.physics.oneloopdiv package.
where $\phi_i$ denotes a "background field" and Latin letters denotes the whole set of its indices\(^1\). So, the main quantity, which determine the effective action is a differential operator

$$D_{ij}^j = \frac{\delta^2 S}{\delta \phi_i \delta \phi_j}. \quad (5)$$

In the most general case, this operator has the following form:

$$D_{ij}^j = \sum_{\mu_1, \mu_2, \ldots, \mu_L}^n K_{\mu_1 \mu_2 \ldots \mu_L} \nabla_{\mu_1} \nabla_{\mu_2} \ldots \nabla_{\mu_L} + \sum_{\mu_1, \mu_2, \ldots, \mu_L}^n S_{\mu_1 \mu_2 \ldots \mu_L} \nabla_{\mu_1} \nabla_{\mu_2} \ldots \nabla_{\mu_L} + \sum_{\mu_1, \mu_2, \ldots, \mu_L}^n W_{\mu_1 \mu_2 \ldots \mu_L} \nabla_{\mu_1} \nabla_{\mu_2} \ldots \nabla_{\mu_L} + \sum_{\mu_1, \mu_2, \ldots, \mu_L}^n N_{\mu_1 \mu_2 \ldots \mu_L} \nabla_{\mu_1} \nabla_{\mu_2} \ldots \nabla_{\mu_L} + \ldots, \quad (6)$$

where $\nabla_\mu$ is a covariant derivative with respect to space-time and gauge indices:

$$\nabla_\mu = \partial_\mu + \Gamma^\alpha_{\mu \gamma} T^\gamma_{\mu i} - \omega_{\mu i}^k T^{\mu k},$$

where $\Gamma^\alpha_{\mu \gamma}$ is a Christoffel symbol and $\omega_{\mu i}^j$ is a connection on the principal bundle. Commuting covariant derivatives it is always possible to make tensors $K, S, W, N, M$ symmetric in the Greek indexes and we shall assume this condition in the further reading.

It also required to introduce the following quantities:

$$(Kn)^i_j = K^{\mu_1 \mu_2 \ldots \mu_L} \nabla_{\mu_1} \nabla_{\mu_2} \ldots \nabla_{\mu_L}, \quad (Kn)^{-1} i_j = \delta^i_k, \quad (Kn)^{-1} i_j (Kn) j_k = \delta^i_k, \quad (Kn) i_j = n_\mu \Phi_i = \partial_\mu \Phi_i + \omega_{\mu i}^j \Phi_j,$$

where $n_\mu$ is a unit vector. The second equation defines tensor $(Kn)^{-1}$ inverse to tensor $Kn$, which is an input tensor for the algorithm\(^2\). The other required input is a curvature tensor with respect to the principal bundle:

$$[\nabla_\mu, \nabla_\nu] \Phi_i = F_{\mu \nu i} \Phi_j \quad (9)$$

Given a set of tensors $K, W, M, (Kn)^{-1}$ and $F$ as input data, Redberry allows to calculate counterterms for the general second and fourth order operators of the following form:

$$D^{(2)}_{ij}^j = K^{\mu \nu} \nabla_\mu \nabla_\nu + W^{\mu \nu} \nabla_\mu \nabla_\nu + M^{\mu \nu} \nabla_\mu \nabla_\nu \quad (10)$$

$$D^{(4)}_{ij}^j = K^{\nu \alpha \beta} \nabla_\nu \nabla_\alpha \nabla_\beta + W^{\nu \alpha \beta} \nabla_\nu \nabla_\alpha \nabla_\beta + M^{\nu \alpha \beta} \nabla_\nu \nabla_\alpha \nabla_\beta \quad (11)$$

Let us illustrate the usage by the particular examples.

### 4.2.1 Non-minimal vector field

Quantum theory of non-minimal vector field $A_\mu$ in a curved space-time is determined from the following action:

$$S = \int d^4x \sqrt{-g} \left( -\frac{1}{4} (\nabla_\mu A_\nu - \nabla_\nu A_\mu)^2 - \frac{1}{2} (\nabla_\mu A^\mu)^2 + \frac{1}{2} P^{\alpha \beta} A_\alpha A_\beta \right)$$

---

\(^1\)Here the actual type of these indices is not important (e.g. this set can contain both space-time and SU(N) indices), so Latin letters used only for convenience.

\(^2\)Redberry also provides facilities to find out the inverse tensor from the given equation. See \texttt{cc.redberry.physics.utils.InverseTensor} class for examples.
where tensor $P_{\alpha\beta}$ is symmetric. One-loop counterterms for this theory were found in [12] and [13]. In the case of vector field, the field indices marked with Latin letters in (5) are space-time indices, so the curvature tensor introduced in (9) is equivalent to Riemann tensor:

$$[\nabla_\mu, \nabla_\nu]A_\alpha = R_{\mu\nu\alpha}^\beta A_\beta$$

The differential operator (5) corresponding to the vector field is:

$$D_\alpha^\beta = \frac{\delta^2 S}{\delta A^\alpha \delta A_\beta} = \delta_\alpha^\beta \Box - \lambda \nabla_\alpha \nabla^\beta + P_\alpha^\beta,$$

where $\Box = g^{\mu\nu} \nabla_\mu \nabla_\nu$ and $\lambda = 1 + 1/\xi$. This is a second order operator, and in order to rewrite it in the form (10), it is necessary to symmetrize the second term by commutation of the covariant derivatives:

$$D_\alpha^\beta = \left( g^{\mu\nu} \delta_\alpha^\beta - \frac{\lambda}{2} \left( g^{\mu\beta} \delta_\alpha^\nu + g^{\nu\beta} \delta_\alpha^\mu \right) \right) \nabla_\mu \nabla_\nu + P_\alpha^\beta + \frac{\lambda}{2} R_\alpha^\beta,$$

where $R_\alpha^\beta$ is the Ricci tensor.

Finally, using equations (7) and (8) it can be easily found that

$$(Kn)_\alpha^\beta = \delta_\beta^\alpha - \lambda n_\alpha n^\beta$$

$$(Kn)^{-1}_\alpha^\beta = \delta_\beta^\alpha + \frac{\lambda}{1 - \lambda} n_\alpha n^\beta$$

Hereby, at this point we have the whole set of the inputting data which is required by the algorithm:

$$F_{\mu\nu\alpha}^\beta = R_{\mu\nu\alpha}^\beta$$

$$K_{\mu\nu\alpha}^\beta = g^{\mu\nu} \delta_\alpha^\beta - \frac{\lambda}{2} \left( g^{\mu\beta} \delta_\alpha^\nu + g^{\nu\beta} \delta_\alpha^\mu \right)$$

$$S_{\mu\alpha}^\beta = 0$$

$$W_\alpha^\beta = P_\alpha^\beta + \frac{\lambda}{2} R_\alpha^\beta$$

$$(Kn)^{-1}_{\alpha\beta} = \delta_\beta^\alpha + \frac{\lambda}{1 - \lambda} n_\alpha n^\beta$$

In further calculations we shall use the definition $\lambda = \gamma/(1+\gamma)$ for convenience (so $\gamma = \lambda/(1-\lambda)$).

The following code calculates one-loop counterterms of the vector field theory in curved space-time (here $\gamma$ used for $\gamma$):

Listing 5 : One-loop counterterms of the theory of vector field.

```python
1 setSymmetric('P_\mu\nu')

3 def KINV = ('KINV_\alpha^\beta = ' +
4 'd_\alpha^\beta + g*n_\alpha*n^\beta').t

6 def K = ('K_\mu\nu_\alpha^\beta = ' +
7 'g_\mu\nu*\alpha\beta - g/(2*(1+g))*(' +
8 'g_\mu\nu*\alpha\beta - g/(2*(1+g))*(' +
9 'g_\mu\nu*\alpha\beta - g/(2*(1+g))*(' +
11 def S = 'S^\rho^\mu_\nu = 0'.t
```

31
```python
def W = ('W^\alpha_\beta = ' +
      'P^\alpha_\beta+g/(2*(1+g))\ast R^\alpha_\beta').t

def F = 'F_\mu\nu\alpha\beta = R_\mu\nu\alpha\beta'.t

// calculates one-loop counterterms of the second order operator
def div = oneloopdiv2(KINV, K, S, W, F)

def counterterms = Factor >> div.counterterms

counterterms = 'P^\alpha_\alpha = P'.t >> counterterms

println counterterms

counterterms = (1/120)*(-32+5*g**2+10*g)*R^\epsilon_{\mu}*R_\epsilon^{\mu}+(1/48)*g**2*P**2+(1/240)*R**2*(28+5*g**2+20*g)+(1/24)*(g**2+12+6*g)*P^\beta_{\alpha}*P^\alpha_{\beta}+(1/12)*g*(4+g)*R_{\nu}^\epsilon*P^\nu_{\epsilon} +(1/24)*R*(g**2+4+2*g)*P
```

In order to obtain one-loop counterterms in the dimensional regularization, one should multiply the result produced by Redberry by \(1/16\pi(d-4)\) and integrate it over the space-time volume:

\[
\Gamma^{(1)} = \frac{1}{16\pi(d-4)} \int d^4x \sqrt{-g} \left( \frac{1}{120}(-32+5\gamma^2+10\gamma)R_{\epsilon\mu}R^{\epsilon\mu} + \frac{1}{48}\gamma^2P^2 + \frac{1}{240}R^2(28+5\gamma^2+20\gamma) + \frac{1}{24}(\gamma^2+12+6\gamma)P_{\beta\alpha}P^{\alpha\beta} + \frac{1}{12}\gamma(4+\gamma)R_{\nu\epsilon}P^{\nu\epsilon} + \frac{1}{24}R(\gamma^2+4+2\gamma)P \right)
\]

The above code is clear enough, but some remarks are needed. First of all, the main method `oneloopdiv2(...)`, which calculates the counterterms of the second order operator, returns a special object, which holds some intermediate results (like e.g. \(RR, RF, FF\) parts of effective action from [9]). The whole result can be obtained by getting the value of `.counterterms` property. All input expressions must be in the same notation as in the original work [9] except tensor \((Kn)^{-1}\), which should be denoted as \(KINV\). At this moment, the implementation requires that all indices of input tensors should be lower Greek indices. Also, it assumed that field indices are placed at the end, so, for example, the first two indices of tensor \(K_{\mu\nu}^{\alpha\beta}\) are contracted with covariant derivatives in (6), while the last two indices corresponds to the indices of vector field. Redberry does not support nonzero tensor \(S\) from (6), however it should be specified explicitly, like it is done in line 11.

**4.2.2 Minimal fourth order operator**

As an example of the fourth order calculation, let us consider minimal fourth order operator of the form

\[
D_{\epsilon}^j = \delta_{\epsilon}^j \Box^2 + W^{\mu\nu} j^j \nabla_\mu \nabla_\nu + M_{\epsilon}^j
\]  
(12)
After symmetrization of the first term, we obtain the following input:

\[ K^{\mu\nu\gamma\delta}_{\alpha\beta} = (1/3) \delta^{\alpha\beta} (g^{\mu\nu} g^{\gamma\delta} + g^{\mu\gamma} g^{\nu\delta} + g^{\mu\delta} g^{\nu\gamma}) \]

\[ S^{\mu\nu\rho\sigma}_{\alpha\beta} = 0 \]

\[ W^{\mu\nu}_{\alpha\beta} = W^{\mu\nu}_{\alpha\beta} \]

\[ N^{\rho}_{\alpha\beta} = 0 \]

\[ M_{\alpha\beta} = M_{\alpha\beta} \]

\[ F_{\mu\nu\alpha\beta} = F_{\mu\nu\alpha\beta} \]

\[ (Kn)^{-1}_{\alpha\beta} \delta^{\alpha\beta} \]

The following code calculates one-loop counterterms for this operator:

Listing 6: One-loop counterterms for the minimal fourth order operator

```python
1 def KINV = 'KINV_\alpha^\beta = d_\alpha^\beta'.t
2 def K = ('K^\mu\nu\gamma\delta_\alpha^\beta = ' +
   'd_\alpha^\beta*1/3*(' +
   'g^\mu\nu*g^\gamma\delta ' +
   '+ g^\mu\gamma*g^\nu\delta ' +
   '+ g^\mu\delta*g^\nu\gamma)' +
   ''').t
3 def S = 'S^\mu\nu\rho\alpha\beta = 0'.t
4 addSymmetry('W_\mu\nu\alpha\beta', 1, 0, 2, 3)
5 def W = 'W^\mu\nu_\alpha^\beta = W^\mu\nu_\alpha^\beta'.t
6 def N = 'N^\rho\alpha\beta = 0'.t
7 def M = 'M_\alpha^\beta = M_\alpha^\beta'.t
8 def F = 'F_\mu\nu\alpha\beta = F_\mu\nu\alpha\beta'.t
9 def div = oneloopdiv4(KINV, K, S, W, N, M, F)
10 def counterterms = EliminateFromSymmetries >> div.counterterms
11 counterterms = 'M^\mu_\mu = M'.t >> counterterms
12 counterterms = 'W_\mu\nu^\alpha_\alpha = W_\mu\nu'.t >> counterterms
13 counterterms = 'W^\alpha_\alpha = W'.t >> counterterms
14 println counterterms
```

```plaintext
33
```

\[ \text{counterterms} = -M+(2/3)*F_{(\nu\beta)}(\epsilon_{\rho_5})*F^{\nu\beta}(\epsilon_{\rho_5})+(32/135)*R_{(\mu\nu)*R_{(\mu\nu)}}+(44/135)*R_{(\mu\nu)}*R^{\mu\nu}_{(\mu\nu)}+(1/9)*R_{(\mu\nu)}*W_{\mu\nu}+(1/9)*R_{(\mu\nu)}*W^{(\mu\nu)}+(1/24)*W^{(\epsilon\delta\alpha)}(\rho_5)^{\epsilon\delta\alpha_{\rho_5}}+(1/48)*W_{\delta\alpha_{\rho_5}}*W_{\delta\alpha_{\rho_5}}(\delta_{\rho_5})\]

--

33
Multiplying this result by \( 1/16\pi(d - 4) \) and integrating over the space-time volume gives:

\[
\Gamma^{(1)} = \frac{1}{16\pi(d - 4)} \int d^4x \sqrt{-g} \left( -M + \frac{2}{3} F_{\nu\beta}^\epsilon \epsilon^{\nu\beta} \right) \\
- \frac{32}{135} R_{\mu\nu} R^{\mu\nu} + \frac{44}{135} R^2 + \frac{1}{9} RW - \frac{1}{9} R_{\mu\nu} W^{\mu\nu} \\
+ \frac{1}{24} W^\alpha{}_{a}^{\rho} W_a{}^{\rho} + \frac{1}{48} W^\alpha{}_{a}^{\rho} W_a{}^{\rho} \right)
\]

where the notation \( M^\mu = M, W^\alpha{}_{\mu\nu} = W^\mu{}_{\nu} \) and \( W^\alpha{}_{\mu} = W \) used. As in the case of the second order operator, it is necessary to explicitly specify, that tensors \( S \) and \( N \) from (6) are equal to zero.

5 Basic internal architecture

5.1 Mappings of indices

Perhaps the most significant difference between tensor- and symbol-oriented computer algebra systems lies in the comparison of mathematical expressions. In the symbol-oriented CASs the result of atomic comparison problem\(^1\) is just a logical true or false, while in the case of tensor-oriented CAS it transforms into a complex pattern matching problem, which produces a complex object as a result. The comparison problem of tensorial expressions will be revealed in this section through several examples.

The most common question, which can be asked about two expressions, is whether they define the same tensor (to within a free indices relabelling). This question arises in such frequent routines like substitutions and reduction of similar terms. Consider the following expressions:

\[ F_{ab} G^{bc} \longrightarrow F_{ij} G^{ij} \]

These two expressions have the same tensorial structure. If rename \( a \) to \( i \) and \( c \) to \( j \) in the left expression one will get exactly the same tensor as defined by the right expression. So, the result of such comparison is not just true or false, but a mapping of free indices of one expression onto free indices of another expression.

One of the major quirks of the problem lies in the fact that free and dummy indices hold completely different places. Mappings of free indices are global for expression, while dummy indices have their scopes. If some free index is present in several places, then its mapping will be the same everywhere. Consider the following two sums:

\[ F_{ab} G^{bc} c + M_{ad} N^{d} c \longrightarrow F_{ij} G^{ij} j + M_{ij} N^{ij} \]

Here \( a \) and \( c \) should be renamed into \( i \) and \( j \) respectively in both summands in order to transform l.h.s. into the r.h.s. If such renaming does not exist, then two expressions define different tensors:

\[ F_{ab} G^{bc} c + M_{ad} N^{d} c \nleftrightarrow F_{ij} G^{ij} j + M_{ij} N^{ij} \]

On the other hand dummy indices have more sophisticated mapping rules, since their explicit names are not important (consider indices \( b \) and \( d \) in the l.h.s and \( q \) in the r.h.s.). So, dummy indices have their scopes, which introduces additional complexity into the problem.

The formal definition of mapping of indices can be formulated as follows: the mapping of indices of tensor A to tensor B is a reflection between free indices of tensor A and indices of

---

\(^1\) Determination of whether two expressions are equal, i.e. operation that is the main building block of such complex routines as pattern matching
tensor $B$ (need not be free), such that if we rename indices of $A$ according to it, then $A$ becomes exactly the same tensor as $B$ (to within dummy indices relabelling).

In fact, mapping of indices is the most frequently executed "atomic" operation in any real calculation. The implementation of mapping procedure and its underlying abstraction is a key achievement of Redberry.

5.1.1 Multiple mappings and symmetries of tensors

In general, several mappings of indices can exist for a pair of tensors. Consider the following primitive example:

$$Z_{ij} + Z_{ji} \rightarrow Z_{ab} + Z_{ba}$$

As one can see, there are two possible mappings:

$$\mathcal{M}_1 = \{ i \rightarrow a, j \rightarrow b \} \quad \text{and} \quad \mathcal{M}_2 = \{ i \rightarrow b, j \rightarrow a \}.$$ 

It is clear that mapping of tensor onto itself gives permutational symmetries of its indices. So in case of the above primitive example, one can find that:

$$Z_{ij} + Z_{ji} \rightarrow Z_{ij} + Z_{ji}$$

gives

$$\mathcal{M}_1 = \{ i \rightarrow i, j \rightarrow j \} \quad \text{and} \quad \mathcal{M}_2 = \{ i \rightarrow j, j \rightarrow i \},$$

where $\mathcal{M}_1$ corresponds to identity symmetry, while $\mathcal{M}_2$ means that if $T_{ij} = Z_{ij} + Z_{ji}$, then $T_{ij} = T_{ji}$.

5.1.2 Sign of mapping

Mappings have a sign property:

$$Z_{ij} - Z_{ji} \rightarrow Z_{ab} - Z_{ba}$$

gives

$$\mathcal{M}_1 = + \left\{ \begin{array}{c} i \rightarrow a \\ j \rightarrow b \end{array} \right\} \quad \text{and} \quad \mathcal{M}_2 = - \left\{ \begin{array}{c} i \rightarrow b \\ j \rightarrow a \end{array} \right\}.$$ 

Negative sign of mapping $\mathcal{M}_2$ means that in order to obtain the r.h.s., one needs to apply mapping to the l.h.s. and negate it. This is an important addition to the definition given in the beginning of this section.

Consider a more complex example. Suppose that tensor $R_{ab}$ is antisymmetric, then:

$$R_{ab}A_c + R_{bc}A_a \rightarrow R_{ij}A_k + R_{jk}A_i$$

gives

$$\mathcal{M}_1 = + \left\{ \begin{array}{c} a \rightarrow i \\ b \rightarrow j \\ c \rightarrow k \end{array} \right\} \quad \text{and} \quad \mathcal{M}_2 = - \left\{ \begin{array}{c} a \rightarrow k \\ b \rightarrow j \\ c \rightarrow i \end{array} \right\}.$$ 

Sign property of mappings and processing both symmetries and anti-symmetries in a common way makes these entities in Redberry fully consistent with each other.
5.1.3 Mappings and substitutions

Usually mappings are used inside a complex routines. In most cases mapping rules are calculated to be applied to some tensors. Common pattern of mappings usage can be illustrated by their application in substitutions. Suppose one need to apply the following substitution rule:

$$F_{\alpha\beta} = R^\mu_{\alpha\beta\mu}$$

to the expression:

$$F_{\rho\tau} G^{\mu\tau}.$$  

In order to do this, one need to perform the following steps:

1. Find the l.h.s. of the substitution in the expression: $F_{\rho\tau} G^{\mu\tau}$
2. Perform the comparison, i.e. build mapping of indices:

$$F_{\alpha\beta} \rightarrow F_{\rho\tau} = \begin{cases} \alpha \rightarrow \rho \\ \beta \rightarrow \tau \end{cases}$$

3. Check for dummy indices clash and resolve it. Here tensor $R^\mu_{\alpha\beta\mu}$ conflicts with $F_{\rho\tau} G^{\mu\tau}$, so we can resolve problem by adding additional mapping rule:

$$\begin{cases} \alpha \rightarrow \rho \\ \beta \rightarrow \tau \\ \mu \rightarrow \alpha \end{cases}$$

4. Apply mapping to the r.h.s. of the substitution:

$$R^\mu_{\alpha\beta\mu} \rightarrow R^\alpha_{\rho\tau\alpha}$$

5. Replace:

$$F_{\rho\tau} G^{\mu\tau} \rightarrow R^\alpha_{\rho\tau\alpha} G^{\mu\tau}$$

5.1.4 Examples and performance

Comparison of mathematical expressions is the most frequent atomic operation in the program and its performance directly affects execution time of any calculation. Because of this, its implementation is one of the main targets for optimization.

The first remarkable thing concerning performance, is that in some cases (e.g. when tensors have a lot of symmetries etc.) the number of all possible mappings can be huge (about $N!$, where $N$ is a number of free indices). However, in most cases there is no need to compute all possible mappings. Thus, Redberry uses iterator pattern to enumerate possible mappings, and the calculation of each subsequent mapping occurs only on corresponding step of iteration. Consider example:

```java
1 setAntiSymmetric(‘R_ab’)
2 def from = ‘R_(ab)*A_c + R_(bc)*A_a’.
t,
3 to = ‘R_(ij)*A_k + R_(jk)*A_i’.
t
4 def mappings = from % to
5 mappings.each { println it }
```

```java
> + { _a -> _i, _b -> _j, _c -> _k } 
> - { _a -> _k, _b -> _j, _c -> _i }
```
Here, `mappings` object is an iterator\(^1\) over possible mappings from tensor \(\text{from}\) to tensor \(\text{to}\), which calculates mappings only on demand. It has the main method `take()` (\(++\) in Groovy syntax) that calculates and returns the next mapping or \(\text{null}\) if no more mappings exists.

Consider the following examples:

```plaintext
1. \(\text{def from = 'A_m^n'.t, to = 'A^a_b'.t}\)
   (from % to).each {
     println it
     ▶ +{_m -> ^a, ^n -> _b}
   }

2. \(\text{addSymmetry('R_{abcd}', 0, 2, 1, 3)}\)
   from = 'R_a^bi_l'.t
   to = 'R_{aj}^b^j'.t
   (from % to).each {
     println it
     assert to == (it >> from) }
     ▶ +{_a -> _a, ^b -> ^b, ^i -> _j, _l -> ^j}
     ▶ +{_a -> _a, ^b -> _j, ^i -> ^b, _l -> ^j}

3. \(\text{setAntiSymmetric('A_{mn}')\)}
4. \(\text{setAntiSymmetric('F_{mnab}')\)}
5. \(\text{from = '(A_m^n - A_m^p*A_p^n)*F_{nk}^i_j + A_{mn}A^n_j*A^i_k'.t}\)
   to = '-(A_d^a + A_p^a*A_d^p)*F^d_kq^i - A^a_b*A^b_q*A^i_k'.t
   (from % to).each {
     println it
     assert to == (it >> from) }
     ▶ +{^i -> _k, _j -> _q, _k -> ^i, _m -> ^a}
     ▶ -{^i -> ^i, _j -> _q, _k -> _k, _m -> ^a}
```

As it seen, it is possible to map indices with different states, i.e. upper indices onto lower and vice-versa (for metric types only). In order to apply mapping rules to tensor one can use \(\text{>>}\) operator (as in lines 4, 10, 18). This will automatically perform raising or lowering of indices if it is meant by mapping and resolve dummy clashes.

### 5.2 Graph theory and tensors

As was mentioned in the previous section, finding of possible mappings between indices of tensors is a computationally hard problem. One of the main ideas, used in the mappings module in Redberry, is that contractions of indices define a mathematical graph.

For simplicity, let’s assume anywhere where it is not specified, that all tensors in this section are completely symmetric. Here is a formal description of representation of a product as a mathematical graph:

- Each multiplier matches a vertex of the graph which label is equal to tensor name (see below)
- Each pair of dummy indices matches an edge of the graph

\(^1\)It implements an `OutputPortUninterruptible` pattern from `redberry-pipe` library for concurrent algorithms.
• Each free index matches a special vertex, labeled ‘—’, and connected to the corresponding multiplier. So, free indices are in some sense contracted with abstract multipliers.

Here, term "tensor name" is used to distinguish the mathematical nature of different tensors. In case of simple tensors two tensors are considered to have different names if and only if they have different mathematical nature. For example, all of the following tensors

\[ A_{\mu\nu}, \quad A_\alpha^\beta, \quad A_\alpha^\alpha \]

are considered to have equal names in Redberry, but, for example, names of tensors \( B_{\mu\nu} \) or \( A_{\alpha\beta\gamma} \) differs from those listed. Internally, Redberry represents simple tensor names as integer numbers and assigns unique integers for simple tensors with different mathematical nature. If some multiplier is a complex tensor (e.g. sum), then Redberry uses its hash code as a label for graph vertex.

In order to illustrate proposed rules, let’s consider the following tensor:

\[ A_\alpha^\mu A_{\mu\beta} C_{\rho\nu} \]

This tensor has two free indices \( \alpha \) and \( \beta \), and since all of the multipliers are completely symmetric, it is easy to show that this tensor is also symmetric:

\[ A_\alpha^\mu A_{\mu\beta} C_{\rho\nu} = A_\beta^\mu A_{\mu\alpha} C_{\rho\nu}. \]  (13)

Fig. (1a) and (1b) illustrate graphs that represent the l.h.s. and r.h.s. of the above equality. Since these tensors are equal, they have the same graph representation, which is shown in Fig. (1c).

Figure 1: Graph representation of tensors \( A_\alpha^\mu A_{\mu\beta} C_{\rho\nu} \) and \( A_\beta^\mu A_{\mu\alpha} C_{\rho\nu} \). Figures (a) and (b) show correspondence between graph edges and indices of tensors. Figure (c) shows the common graph representation.

It is clear that symmetry (13) leads to a nontrivial automorphism of the corresponding graph which is shown in Fig. 2. On the other hand, nontrivial graph automorphisms, which permute free indices (‘—’ vertices), are equivalent to tensor symmetries.
Figure 2: Automorphism of the tensor graph $A^{\mu \nu} A_{\mu \beta} B C_{\rho \nu}$, which proofs its symmetry properties.

The above example gives a good idea of the tight connection between the tensor-oriented computer algebra and computational graph theory. Many problems specific to tensor-oriented CAS can be re-formulated or reduced to the known problems in the graph theory and vice-versa. It is now clear that mappings problem is closely related with graph isomorphism problem.

In order to take into account particular permutational symmetries of simple tensors, Redberry performs some kind of an additional edge labeling. So, in general case of arbitrary tensor, we have disconnected, undirected graph with labeled vertices and edges. Graph problems, such as effective finding of isomorphisms or automorphisms of graphs are one of the central problems in computer science, and actively studied for decades (see [14]). Many algorithms were proposed to solve these problems, and while many of them seem to perform well on random graphs, a major drawback of these algorithms is their exponential time performance in worst cases. Moreover, it is still an open question: whether graph isomorphism (or automorphism) is $P$ or $NP$-complete.

Additionally to the graph isomorphism problem, more complex problems arise when building mappings between tensors, which maps free indices onto dummy indices. Consider the following example. Suppose we need to build a mapping from tensor $F$ to tensor $T$, where

from tensor $F$: $A_{\mu \alpha \nu} A_{\beta \mu} A_{\delta \sigma} A_{\nu \gamma} A_{\xi} A_{\xi} A_{\lambda} A_{\xi} A_{\xi} A_{\rho \sigma} A_{\rho \tau}$

to tensor $T$: $A_{\xi \alpha \phi} A_{\beta \phi} A_{\chi \sigma} A_{\psi \chi} A_{\lambda \psi} A_{\psi} A_{\xi} A_{\xi} A_{\psi} A_{\rho \sigma} A_{\rho \tau}$

Tensor $F$ have two free indices $\xi$ and $\lambda$, while tensor $T$ is a scalar. One can see, that mapping should simply contract free indices of $F$, like

$$+ \begin{cases} \xi \rightarrow \xi \\ \lambda \rightarrow \xi \end{cases}$$

Nevertheless, this simple answer involves building a special kind of mapping between the corresponding graphs, which are shown in Fig. 3. We see, that this mapping is not an exact isomorphism, but a surjection. It is clear, that calculation of such mappings implies finding of isomorphic structures in corresponding tensors, and can be effectively reduced to the graph isomorphism problem.

Another computationally hard problem arises when applying substitution rules with a product in the l.h.s. Suppose one have a substitution rule $A \rightarrow B$, where $A$ is a product of tensors, and one needs to apply it to tensor $C$. Then one needs to find all subproducts of $C$, such that there is mapping from it to $A$. This implies, that one needs to find all subgraphs of graph $C$ which are isomorphic to graph $A$. This is a well-known $NP$-complete subgraph isomorphism problem [15], and algorithms known to date, in general, have an exponential or even factorial execution time in the worst case.
Redberry has internal implementation of graph and subgraph isomorphism problems. The implementation is based on the topological hashing and hash refinement mixed with depth-first search. It actively utilizes additional information about symmetries of tensors, and the worst case occurs only when there are a lot of similar symmetric multipliers in the target products. While, the algorithms developed in Redberry have a pretty good performance in the most cases arises in physical problems, they still have bad performance in some cases, so improvement of these algorithms is one of the main goals of current development of Redberry.

5.3 Expression-tree traversal and modification

As mentioned in Sec. 2.2 each tensor in Redberry is a container of its child tensors, so any complex expression becomes a hierarchical tree of such containers. Iteration over direct child elements of a tensor described in Sec. 2.2. Besides, there are a special tools for iteration and modification over a whole tree.

There is a core class\(^1\) in Redberry that performs traversal over any given expression. It basically generates a sequence of traversal events like: entering or leaving of subexpressions. The main feature of this class is its ability to in-place modify a tree while traversing. Redberry provides simple facade classes for tree traversal. There are two basic ways to perform a depth-first search on expression tree:

```python
1 def t = 'a + Sin[x + y]' \t
2 t.parentAfterChild { a -> print a.toString() + ', ' }
```

\(\text{TreeTraverseIterator in package } cc.redberry.core.tensor.iterator\)

---

1 TreeTraverseIterator in package cc.redberry.core.tensor.iterator
This example illustrates steps of parent-after-child and parent-before-child iteration modes. It is also possible to specify a particular guide in order to restrict iteration only to a parts of expression:

```python
t = 'a*b + c + Sin[x + y]' 

def guide = { 
    tensor, parent, positionInParent -> 
    if (tensor.class == Sin) ShowButNotEnter 
    else if (parent == 'a*b'.t) DontShow 
    else Enter 
} as TraverseGuide 

t.parentAfterChild (guide) { a -> print a.toString() + ', ' }
```

The guide takes three arguments: current tensor, its parent tensor and integer position in parent tensor and returns TraversePermission. In such a way guide tells the iterator to step over current tensor (DontShow), or show current tensor, but not to go deeper into its children (ShowButNotEnter), or to continue normal iteration (Enter). The guide shown above guides the iteration as follows: if current tensor is of type Sin, then it will be shown during iteration, but the iterator will not go inside it; if parent of the current tensor is equal to a*b, then such tensor will not be shown at all; otherwise iterator will continue normal iteration.

Modification of expression tree can be performed in the same way. As an example, lets consider a naive implementation of a substitution transformation:

```python
def subs = { 
    t, expr -> 
    t.transformParentAfterChild { 
        def c = ++(expr[0] % it) 
        if (c != null) c >> expr[1] 
        else it 
    } 
} 

def t = 'z_m*Cos[x_m*y^m - x_n*(z^n + t^n)] + t_m'.t 
println subs(t, 'z_a + t_a = y_a'.t) 
```

\(^1\)one should import Sin from cc.redberr.core.tensor.functions, TraverseGuide from cc.redberr.core.iterator and also statically import TraversePermission from the last package.
The key aspect of parent-after-child modification, is that if child node is changed, then its parent node will be reduced to standard form (see Sec. 2.4) before it will be shown\(^1\).

Another important thing concerning expression tree modification, is that any conflicting dummy indices will be relabeled automatically:

\[
\begin{align*}
& t = 't_a*(x + t_a*t^a*x)' \cdot t \\
& println \ subs(t, 'x = F_a^a' \cdot t)
\end{align*}
\]

\[\downarrow\] \( t_{a}*(F_{b}^{b} + t_{b}*t^{b}*F_{c}^{c}) \)

6 Conclusions

In this paper we presented Redberry — an open source computer algebra system designed to manipulate with symbolic tensorial expressions. Redberry is a computer algebra system, which considers both tensors and indexless expressions in a common way. It provides basic tensor-specific routines such as tensor symmetries, multiple index types, dummy indices handling, \LaTeX-style I/O, mappings of tensor indices and a set of tensor-specific transformations etc. Rich functionality of Redberry was demonstrated on a complex physical problems: Feynman graphs and one-loop counterterms calculations. Compared to many other tensor-oriented CASs, Redberry provides a simple and convenient facade to modern high-level programming language (Groovy), which makes it possible to use all features of general-purpose programming language combined with domain-specific features required by computer algebra.

At the moment, Redberry does not implement many features, which should be implemented in a general-purpose tensor-oriented CAS. However, the architecture of Redberry is built to be open for further extensive development. This is a first public release of Redberry CAS, and the development is ongoing. We currently working under the following issues: pattern matching, derivatives, expressions standard form improvement, performance of symmetries, mappings, and factorization.

Redberry code is well covered by over 800 unit tests. However, there are several known bugs (that not affects correctness of calculations), which are listed in Redberry issue tracker. Redberry performance can be demonstrated by the following numbers\(^2\): the whole calculation from Sec. 4.1 (Compton scattering in QED) takes about 0.5 seconds, calculation of counterterms of non-minimal squared vector field takes less then 3 minutes, non-minimal vector field — about 22 seconds, minimal fourth order operator — about 2 seconds. The calculation of trace of 8 gamma matrices takes less then 20 ms, while, for example, FeynCalc makes it in 300 ms (on the same hardware). The calculation of trace of 12 gamma matrices takes less then 3 seconds in Redberry.

Redberry is licensed under GNU GPLv3. Additional documentation, examples and installation instructions are available at \url{http://redberry.cc}. The source code and issue tracker can be found at \url{http://bitbucket.org/redberry}. We are welcome for contributions and any suggestions.

\(^1\)The first modification will be performed on the sum \( z^n + t^n \), which will be replaced with \( y^n \). After this modification, the whole branch of the expression tree, which contains this term, will be permanently reduced to the standard form. So, when iteration will reach the argument of cosine, which will be \( x_m*y^m - x_n*y^n \), it will be reduced to zero. On the next step of the iteration, it finally becomes possible to replace obtained \( z_m + t_m \) with \( y_m \).

\(^2\)on Intel Core i5 2.27 GHz
Acknowledgments

The authors are grateful to Petr Pronin and Konstantin Stepanyantz for a useful discussion. The work of S.V.P. was financially supported by Russia Foundation for Basic Research (grants #12-02-31249a and #12-02-31408a), grant of the president of Russian Federation (grant #MK-3513.2012.2) and grant of FAIR-Russia Research Center. The work of D.A.B. was financially supported by the Molecular and Cell Biology program of the Russian Academy of Sciences.

A Inputting matrix expressions

Matrices are tensors with a subset of indices marked as matrix indices. These indices have nonmetric types, which means, that raising and lowering are forbidden. The rules of matrix multiplication can be formulated as follows. Suppose we have two tensors

\[ A_{\mu}^{u_1...u_nu_1...u_n} \] \[ B_{\nu}^{\tilde{u}_1...\tilde{u}_kn_1...n_k} \]

where \( u_i \) and \( l_i \) denotes matrix upper and matrix lower indices respectively, and \( \mu \) and \( \nu \) denotes a whole subset of other indices. Then the product \( T_{\mu\nu} = A_{\mu} B_{\nu} \) has the following form:

\[ T_{\mu\nu}^{u_1...u_n\tilde{u}_k1...\tilde{u}_k} = A_{\mu}^{u_1...u_n} B_{\nu}^{c_1...c_n\tilde{u}_k1...\tilde{u}_k} \]

if \( n_k \leq k_u \)

\[ T_{\mu\nu}^{u_1...u_n\tilde{l}_k1...\tilde{l}_k} = A_{\mu}^{u_1...u_n} B_{\nu}^{c_1...c_k\tilde{l}_k1...\tilde{l}_k} \]

if \( n_k > k_u \)

These definitions becomes especially clear in a particular cases. Consider, for example, three matrices: matrix \( v^i \) with one upper matrix index (usually called vector), matrix \( \gamma_{ij} \) with one upper and one lower index, and matrix \( \bar{v}^i \) with one lower matrix index (usually called covector). Then using the above rules, one can obtain the well-known expressions:

\[ T_{\mu\nu} = \gamma_{\mu} \gamma_{\nu} \rightarrow T_{\mu\nu}^{ij} = \gamma_{\mu}^i \gamma_{\nu}^j \] (matrix)

\[ T_{\mu} = \gamma_{\mu} v \rightarrow T_{\mu}^{i} = \gamma_{\mu}^i v^c \] (vector)

\[ T_{\mu} = \bar{v} \gamma_{\mu} \rightarrow T_{\mu}^{i} = \bar{v} \gamma_{\mu}^i \] (covector)

\[ T_{\mu\nu} = \bar{v} \gamma_{\mu} \gamma_{\nu} v \rightarrow T_{\mu\nu}^{ij} = \bar{v} \gamma_{\mu}^i \gamma_{\nu}^c v^j \] (scalar)

Redberry has a tool to specify what tensors should be considered as matrices. It automatically inserts the additional matrix indices according to the formulated rules. Consider the examples:

1 `defineMatrices 'G_a', 'G', Matrix1.matrix,
2 'v', Matrix1.vector, 'cv', Matrix1.covector`

4 `println 'G_a*G_b'.t`
   \( \triangleright \) \( G_a^a\_b' * G_b^b' \)

5 `println 'G_a*v'.t`
   \( \triangleright \) \( G_a^a\_b' * v^b \)

6 `println 'cv*G_a'.t`
   \( \triangleright \) \( cv_a\_a' + G_a^a\_b' \)

7 `println 'G*G_a = G*v*cv*G_a + f_a'.t`
   \( \triangleright \) \( G^a\_c' * G_a^c\_b' = G^a\_c' * v^c' * cv_d' * G_a^d' \_b' + f_a^d' \_c' \_b' \)
8  println 'cv*G_a*G_b*v + g_ab'.t
   ▷ v_a'*G_a^a'_b'*v^b' + g_ab
9  println 'Tr[G_a*G_b + n_b*G_a] + n_a*n_b'.t
   ▷ G_a^a'_b'*G_b^b'_a' + n_b*G_a^a'_a' + n_a*n_b
10 println 'Tr[G_a*G_b + n_b*G_a + n_a*n_b]'.t
    ▷ G_a^a'_b'*G_b^b'_a' + n_b*G_a^a'_a' + n_a*n_b*d^a'_a'

In the first two lines we specify tensors which will be considered as matrices in further calculations. Method `defineMatrices` takes a set of string representations of matrices, where the corresponding matrix signature (type and number of upper/lower matrix indices) is specified after each set. By default, Redberry provides four predefined nonmetric types: `Matrix1` (Latin lower case letters with strokes), `Matrix2` (Latin upper case letters with strokes), `Matrix3` (Greek lower case letters with strokes), `Matrix4` (Greek upper case letters with strokes). The number of upper and lower indices of matrix type is specified using the corresponding property: `vector` means one upper index, `covector` means one lower index, `matrix` means one upper and one lower index. In the general case of \( p \) upper and \( q \) lower indices, one can do

11  defineMatrices 'M_{\mu}', Matrix1.tensor(2, 3)
13  println 'G*M_{\alpha}'.t
    ▷ G^{a'}_{f'}*M_{\alpha}^{f'b'}_{c'd'e'}

As we see from the lines 7 and 10, Redberry inserts similar free matrix indices for the l.h.s. and r.h.s. of the expression. Moreover, if, for example, some sum contains both matrix (with non empty free matrix indices) and non-matrix expressions (without matrix indices), then the later will be multiplied by the identity matrices, i.e. Kronecker deltas (compare to line 8).

Redberry provides a special syntax for traces of matrices shown in the lines 9 and 10. By default the trace is taken with respect to all matrix indices, but if some expression is a matrix with respect to several indices types, then it is possible to specify particular types of indices for which the trace should be taken:

1  defineMatrices 'A, B', Matrix1.matrix, Matrix2.matrix
3  println 'Tr[A*B]'.t
   ▷ A^{a'}_{b'}^{A'}_{B'}*B^{b'}_{a'}^{B'}_{A'}
4  println 'Tr[A*B, Matrix1]'.t
   ▷ A^{a'}_{b'}^{A'}_{C'}*B^{b'}_{a'}^{C'}_{B'}

B  Redberry artifacts structure

Redberry project consists of three main artifacts, which are published in Maven Central Repository:
- **cc.redberry.core** — the core framework written in pure Java, which contains the implementation of basic Redberry routines and general-purpose transformations.

- **cc.redberry.physics** — applications of Redberry in real physical problems. It contains tools for Feynman graphs and one-loop counterterms calculation.

- **cc.redberry.groovy** — a Groovy facade for Redberry. It contains the classes and categories\(^1\) defining DSL features and syntax notations for common routines from *redebrry-core* and *redberry-physics*.

The current version of all Redberry artifacts is 1.1. All official releases of Redberry artifacts including binary *jar* files, sources, documentation and tests, can be found in Maven repository. Redberry sources and version control are hosted at bitbucket.org/redberry. Redberry is free and open source project and anyone can contribute to Redberry with new features using Bitbucket forks.

---

\(^1\)see Groovy documentation
References

[1] K. Peeters, “Introducing Cadabra: A Symbolic computer algebra system for field theory problems,” arXiv:hep-th/0701238 [hep-th].

[2] K. Peeters, “Cadabra: a field-theory motivated symbolic computer algebra system,” Comput.Phys.Commun. 176 (Apr., 2007) 550–558, arXiv:cs/0608005.

[3] Mertig, R. and Bohm, M. and Denner, Ansgar, “Feyn Calc: Computer algebraic calculation of Feynman amplitudes,” Comput.Phys.Commun. 64 (1991) 345–359. FeynCalc home page: http://www.feyncalc.org/.

[4] SymPy Development Team, SymPy: Python library for symbolic mathematics, 2010. http://www.sympy.org.

[5] J. Martin-Garcia, “The xAct package,”. xAct home page: http://www.xact.es/.

[6] H. Kredel, “On a java computer algebra system, its performance and applications,” Science of Computer Programming 70 (2008) 185 – 207. Special Issue on Principles and Practices of Programming in Java (PPPJ 2006).

[7] H. Kredel, “Unique Factorization Domains in the Java Computer Algebra System,” in Automated Deduction in Geometry - 7th International Workshop, ADG 2008, Shanghai, China, September 22-24, 2008. Revised Papers, T. Sturm and C. Zengler, eds., vol. 6301 of Lecture Notes in Computer Science, pp. 86–115. Springer, 2008.

[8] M. E. Peskin and D. V. Schroeder, An Introduction to Quantum Field Theory. Perseus Books, Cambridge, Massachusetts, 1995.

[9] P. I. Pronin and K. V. Stepanyantz, “One loop counterterms for the dimensional regularization of arbitrary lagrangians,” Nucl.Phys. B485 (1997) 517–544, hep-th/9605206 [hep-th].

[10] P. Pronin and K. Stepanyantz, “New Tensor Package for REDUCE system,” in New Computation Technique in Physics Research, B.Denby and D.Perred-Gallix, eds., vol. 4. Word Scientific, Singapure, 187, 1995.

[11] P. Pronin and K. Stepanyantz, “One loop counterterms for higher derivative regularized Lagrangians,” Phys.Lett. B414 (1997) 117–122, arXiv:hep-th/9707008 [hep-th].

[12] E. Fradkin and A. A. Tseytlin, “Renormalizable asymptotically free quantum theory of gravity,” Nucl.Phys. B201 (1982) 469–491.

[13] A. Barvinsky and G. Vilkovisky, “The Generalized Schwinger-Dewitt Technique in Gauge Theories and Quantum Gravity,” Phys.Rept. 119 (1985) 1–74.

[14] Wikipedia, The Free Encyclopedia, “Graph isomorphism problem,” http://en.wikipedia.org/wiki/Graph_isomorphism_problem.

[15] Wikipedia, The Free Encyclopedia, “Subgraph isomorphism problem,” http://en.wikipedia.org/wiki/Subgraph_isomorphism_problem.