ISSN 0361-7688, Programming and Computer Software, 2019, Vol. 45, No. 2, pp. 58–64. © Pleiades Publishing, Ltd., 2019.
Russian Text © D.S. Kulyabov, A.V. Korol'kova, L.A. Sevast'yanov, 2019, published in Programmirovanie, 2019, Vol. 45, No. 2.

New Features in the Second Version of the Cadabra Computer Algebra System

D. S. Kulyabov[a,b,*], A. V. Korol'kova[a], and L. A. Sevast'yanov[a,c]

[a] *Department of Applied Probability and Informatics, Peoples' Friendship University of Russia (RUDN University), ul. Miklukho-Maklaya 6, Moscow, 117198 Russia*

[b] *Laboratory of Information Technologies, Joint Institute for Nuclear Research, ul. Zholio-Kyuri 6, Dubna, Moscow oblast, 141980 Russia*

[c] *Bogoliubov Laboratory of Theoretical Physics, Joint Institute for Nuclear Research, ul. Zholio-Kyuri 6, Dubna, Moscow oblast, 141980 Russia*

**e-mail: kulyabov-ds@rudn.ru*

Received September 02, 2018; revised September 02, 2018; accepted September 20, 2018

Abstract—In certain scientific domains, there is a need for tensor operations. To facilitate tensor computations, computer algebra systems are employed. In our research, we have been using Cadabra as the main computer algebra system for several years. Recently, an operable second version of this software was released. In this version, a number of improvements were made that can be regarded as revolutionary ones. The most significant improvements are the implementation of component computations and the change in the ideology of the Cadabra's software mechanism as compared to the first version. This paper provides a brief overview of the key improvements in the Cadabra system.

DOI: 10.1134/S0361768819020063

1. INTRODUCTION

Computer algebra systems that support tensor calculus can be conventionally divided into three groups [1]. The first group includes universal tensor calculus systems designed mainly for use in the general theory of relativity. In these systems, focus is placed on computing characteristic quantities of Riemannian geometry (Christoffel symbols, Riemann tensor) [2–4]. The second group includes computer algebra systems for tensor computations in quantum field theory [5–9]. They focus on computations with Dirac spinors and simple symmetries. The third group of computer algebra systems for tensor calculus is designed mainly as a framework for arbitrary tensor computations. These systems have fewer default templates but provide more expressive means for designing new objects. It is to this group that Cadabra belongs. This paper is devoted to the specific features of this computer algebra system.

Cadabra is currently available in two versions. The first version (hereinafter, Cadabra 1.x) significantly differs from the other computer algebra systems for tensor calculus, particularly, in

- use of the $^{T}_{E}X$ notation;
- convenient definition of tensors of arbitrary types;
- use of Young tableau to describe the symmetry properties of tensors.

However, Cadabra 1.x has significant disadvantages. In particular, this version does not support component computations. Moreover, the monolithic and rigid software structure of the system held no promise for the implementation of component computations in the near future. However, a revolutionary step taken in the second version of the system (hereinafter, Cadabra 2.x), namely, the use of the Python ecosystem, made it possible to solve these problems. In our opinion, it is the combination of component computations and Python ecosystem that constitutes a revolutionary improvement of the Cadabra system.

Unfortunately, the documentation on Cadabra leaves much to be desired. The website of the system (https://cadabra.science) provides only a few examples. More specific information can be found in research papers that may contain possible applications of Cadabra [10–15].

This paper provides a brief overview of the new features in Cadabra 2.x. The paper is organized as follows. Section 2 discusses the implementation of Cadabra 1.x and 2.x. Section 3 describes the process of action variation for a source-free electromagnetic field to illustrate the syntax of Cadabra 2.x. To compare the 1.x and 2.x syntaxes, see [13–15]. Section 4 considers one of the most important features in Cadabra 2.x: its transparent interaction with the universal scalar computer algebra system SymPy. Section 5 describes the

key (in our opinion) innovation in Cadabra 2.x—component computations—by the example of finding basic quantities for the general theory of relativity, namely, Christoffel connection and different curvatures. Finally, Section 6 illustrates work with graphics in Cadabra 2.x.

When implementing the examples, we used code fragment from the Cadabra 2.x documentation (https://cadabra.science/tutorials.html).

2. IMPLEMENTATION FEATURES OF CADABRA 1.X AND 2.X

Each computer algebra system has its own implementation features. There are several levels of implementation:

- notation;
- manipulation language;
- implementation language;
- extension language.

Not every computer algebra system has all these levels. For instance, in most systems, the notation is based on the manipulation language.

In Cadabra 2.x (as in 1.x), the notation is based on the $^{T}_{E}{}^{X}$ notation (more precisely, $^{T}_{E}{}^{X}$-like notation). In this case, a certain subset of $^{T}_{E}{}^{X}$ symbols (letters of different alphabets, symbols for integral and derivatives, etc.) is used. Above all, indices are denoted by the symbols _ and ^, as in the $^{T}_{E}{}^{X}$ system.

The manipulation language is used to work in the system. The syntax of Cadabra 1.x is quite simple and more oriented to code parsing rather than to user-friendliness. This approach to language design was common in the early years of computing technology development (e.g., Shell and Perl languages). Cadabra 2.x takes a qualitative step forward by switching to Python. As a result, all operations in Cadabra 2.x are written in a Python-like syntax.

Cadabra 1.x is implemented in C++ with the use of the LiE computer algebra system [16] (currently, the compilation of the LiE system causes certain difficulties). Its main purpose is processing Lie groups, on which operations with tensor symmetries are based. It should be noted that the entire system was actually implemented by one person. This required great efforts. However, the presence of only one author and the monolithic software structure of the system caused some concern.

In Cadabra 2.x, the author fundamentally changed his approach to the structure of the system by integrating it with the Python ecosystem. The 2.x system is still written in C++, but Python is used as a glue language (and also as a manipulation language). In addition, Python can be employed for writing extensions.

The Python infrastructure opens access to a large number of scientific libraries, including the SciPy project [17]. This allows the SciPy libraries to be used seamlessly, transparently to the user. In our opinion, this allowed Cadabra 2.x to take a revolutionary step forward.

3. ELEMENTS OF THE 2.X SYNTAX

Let us recall some elements of the 1.x and 2.x syntaxes. As an example, we obtain the source-free Maxwell's equation [18]

$$\partial_\mu F^{\mu\nu} = 0, \tag{1}$$

by varying the action

$$S = -\frac{1}{4}\int F_{\mu\nu}F^{\mu\nu}dx. \tag{2}$$

In this case, the Maxwell tensor $F_{\mu\nu}$ is expressed in terms of the vector potential:

$$F_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu. \tag{3}$$

Again, both versions of Cadabra use the $^{T}_{E}{}^{X}$ notation. In Cadabra 2.x, it becomes possible to use functionality of Python, in particular, define functions directly in the text of a program written in Cadabra 2.x. Note that, just like Cadabra 1.x, the 2.x version does not process resulting expressions by default (except for collecting like terms); this processing is left to the user. If it is required to apply a certain set of rules to all expressions used, then the function `post_process` can be called, which is executed after each operation (in fact, `post_process` is simply a Python function):

```
def post_process (ex):
    sort_product (ex)
    canonicalise (ex)
    collect_terms (ex)
```

If necessary, this function can be made empty, thus disabling any processing of expressions.

The interface of Cadabra 2.x exploits the ideology of a notepad, i.e., in addition to writing program code, the user can also add comments. However, in contrast to iPython [19], text is written in the $^{L}_{A}{}^{T}_{E}{}^{X}$ syntax rather than in the Markdown syntax [20].

The object in Cadabra 2.x can be assigned a property, which, in turn, has a set of its own settings. Since we are dealing with tensors, the most useful property is `Indices`. The option `position=free` allows the system to raise and lower indices:

```
{\mu, \nu, \rho}::Indices(position=free).
x::Coordinate.
\partial{#}::Derivative.
```

Here, # is a wildcard. The dot at the end of the expression suppresses the output (as is common in computer algebra systems).

To work with abstract indices, it is necessary to take into account the symmetry properties of tensors. In addition, when differentiating and integrating, the

coordinate dependence of objects needs to be taken into account:

```
F_{\mu\nu} :: AntiSymmetric;
F_{\mu\nu} :: Depends (x).
A_{\mu} :: Depends (x, \partial{#}).
\delta{#} :: Accent;
```

Attached property AntiSymmetric to $F_{\mu\nu}$.

Attached property Accent to δ#.

In this case, the variation sign δ is regarded as a modifier (rather than as an object with the `Derivative` property) and does not introduce any additional computational semantics.

In our example, $F_{\mu\nu}$ is a Maxwell tensor. We express it in terms of the vector potential A_μ (3):

```
F:= F_{\mu\nu} = \partical_{mu}{A_\nu} - \partial_{\nu}{A_{mu}};
```

$$F_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu.$$

Here, the sign := defines a row label (in our case, it is *F*). The label denominates the expression for convenience of referring to it.

Next, the action for the electromagnetic field (2) is defined:

```
S:=-1/4\int{F_{\mu\nu}F^{\mu\nu}}{x};
```

$$-\frac{1}{4}\int F^{\mu\nu}F_{\mu\nu}dx.$$

By substitution, the action is expressed in terms of the vector potential A_μ:

```
substitute (S, F);
```

$$-\frac{1}{4}\int (\partial_\mu A_\nu - \partial_\nu A_\mu)(\partial^\mu A^\nu - \partial^\nu A^\mu)dx.$$

Then, the action has the following form:

```
S;
```

$$-\frac{1}{4}\int (\partial_\mu A_\nu - \partial_\nu A_\mu)(\partial^\mu A^\nu - \partial^\nu A^\mu)dx.$$

The value of the action in this expression differs from the initial one. It takes the form that it received after the last computation. This is a bit unusual. The point is that most computer algebra systems are implemented using functional languages, or they follow a functional paradigm in which variables have the property of immutability. In this case, the label acts as a variable in imperative languages (Python is an imperative language). This makes work in Cadabra 2.x necessary linear: the user cannot randomly navigate through the notebook and perform computations at arbitrary points.

Let us vary the action:

```
vary(S, $A_{\mu}->\delta{A_{mu}}$);
```

$$-\frac{1}{4}\int ((\partial^\mu A^\nu - \partial^\nu A^\mu)(\partial_\mu \delta A_\nu - \partial_\nu \delta A_\mu) + (\partial_\mu A_\nu - \partial_\nu A_\mu)(\partial^\mu \delta A^\nu - \partial^\nu \delta A^\mu))dx.$$

Here, the expression itself, rather than its label, is used. For this purpose, the expression is put between two dollar symbols ($), as in the standard $\TeX$.

Next, we expand the products and collect the like terms:

```
distribute (S);
```

$$-\frac{1}{4}\int (4\partial^\mu A^\nu \partial_\mu \delta A_\nu - 4\partial^\mu A^\nu \partial_\nu \delta A_\mu)dx.$$

Then, integration by parts is carried out:

```
integrate_by_parts(S,$\delta{A_{\mu}}$);
```

$$-\frac{1}{4}\int (-4\delta A^\mu \partial^\nu (\partial_\nu A_\mu) + 4\delta A^\mu \partial^\nu (\partial_\mu A_\nu)dx.$$

In this case, integration by parts is quite a formal action that uses only the `Derivative` property of the object.

Once again, we perform the substitution, expand the products, and collect the like terms:

```
substitute (_, $\partial_{\mu}{A_{\nu}} -> 1/2\partial_{\mu}{A_{\nu}} + 1/2 F_{\mu\nu} + 1/2 \partial_{\nu}{A_{\mu}}$);
```

$$-\frac{1}{4}\int \left(-4\delta A^\mu \partial^\nu \left(\frac{1}{2}\partial_\nu A_\mu - \frac{1}{2}F_{\mu\nu} + \frac{1}{2}\partial_\mu A_\nu\right) + 4\delta A^\mu \partial^\nu \left(\frac{1}{2}\partial_\mu A_\nu + \frac{1}{2}F_{\mu\nu} + \frac{1}{2}\partial_\nu A_\mu\right)\right)dx,$$

```
distribute (_);
```

$$-\int \delta A^\mu \partial^\nu F_{\mu\nu}dx.$$

Here, the label _ denotes the previous expression.

As a result, we obtain the desired Maxwell equation (1):

$$\partial^\nu F_{\mu\nu} = 0. \quad (4)$$

This example demonstrates that the syntax of the manipulation language in Cadabra 2.x is based on the Python syntax, which is more customary than the syntax of the 1.x language.

4. INTERACTION BETWEEN CADABRA AND SYMPY

Computer algebra systems for tensor calculus support quite a small number of operations. They are sufficient for basic manipulations with tensors in the formalism of abstract indices, as well as the index-free formalism. However, in many cases (e.g., full-fledged implementation of component computations), the support of scalar operations is required. If a computer algebra system for tensor calculus is implemented in the framework of a universal computer algebra system,

then no problems arise. However, Cadabra is an independent system. Cadabra 1.x implements a mechanism (though inconvenient) for communication with the Maxima universal computer algebra system; however, it seems that this mechanism was implemented only as a proof of concept.

In Cadabra 2.x, communication with the universal computer algebra system is implemented via SymPy [21]. Moreover, this communication is seamless: the work of the mechanism is invisible for the user, which is owing to implementation of Cadabra 2.x in Python.

Let us illustrate the use of SymPy in Cadabra 2.x by computing the integral

$$\int \frac{1}{x} dx.$$

The main function for the explicit call of SymPy is `map_sympy()`. This function has a side action: it changes the value of the argument. However, as noted above, the absence of immutability is a feature of Cadabra 2.x. Let us consider the simplest call of this function:

```
ex := \int{1/x}{x};
```

$$\int x^{-1} dx$$

```
map_sympy(_);
```

$$\log(x).$$

To confirm the presence of this side action, we check the current value of the expression `ex`:

```
ex;
```

$$\log(x).$$

It can be seen that the value of `ex` has changed.

We can transfer the value of the expression to SymPy and, moreover, call a particular function to process it. For instance, in this case, we can call the SymPy's function `integrate`:

```
ex := 1/x;
```

$$x^{-1}.$$

The second argument of the function is as follows:

```
map_sympy(ex, "integrate");
```

$$\log(x)$$

```
ex;
```

$$\log(x).$$

Again, this confirms the side action of `map_sympy()`.

In Python, the same action can be performed in several ways. Hence, this can be done in Cadabra 2.x. For illustration purposes, let us consider the following variants.

The class method `_sympy_()` can be used as follows:

```
ex := \int{1/x}{x};
```

$$\int x^{-1} dx.$$

While regarding the label ex as an object, we call the method `_sympy_()`:

```
ex._sympy_();
```

$$\log(x).$$

Check the state of the environment:

```
ex;
```

$$\int x^{-1} dx.$$

It is seen that the state of the environment has not changed, i.e., the method `_sympy_()` has no side action.

In addition, we can use the function `sympy` with a method corresponding to a callee function of the SymPy environment:

```
ex := 1/x;
```

$$x^{-1}.$$

Let us call the function `sympy` with the method `integrate`:

```
sympy.integrate(ex);
```

$$\log(x).$$

Note that this function always requires specifying a particular method, which is why it cannot be used in the previous case.

Again, check the state of the environment:

```
ex;
```

$$x^{-1}.$$

It can be seen that the function `sympy` does not have the side action.

Based on the examples considered above, we can conclude that the interaction with SymPy in Cadabra 2.x is implemented in quite an elegant way. In our opinion, the main advantage of this operation is its deep integration with the system, e.g., for implementation of component computations (see Section 5).

5. COMPONENT COMPUTATIONS IN CADABRA 2.X

To illustrate component operations, we find curvature R on a sphere S^2 of radius r:

$$g_{\alpha\beta} = \mathrm{diag}(r^2, r^2 \sin^2 \vartheta).$$

For this purpose, we evaluate Christoffel symbols $\Gamma^{\alpha}_{\mu\nu}$, Riemann tensor $R^{\alpha}_{\beta\mu\nu}$, and Ricci tensor $R_{\alpha\beta}$ [18, 22].

Let us define coordinates and labels for the indices while specifying what values these labels can take:

```
{\theta, \varphi} :: Coordinate;
{\alpha, \beta, \gamma, \delta, \rho,
\sigma, \mu, \nu, \lambda} ::
```

```
  ↪ Indices (values={\varphi, \theta}, position=fixed};
  \partial{#} :: PartialDerivative;
```

Attached property Coordinate to $[\theta, \varphi]$.

Attached property Indices (position=fixed) to $[\alpha, \beta, \gamma, \delta, \rho, \sigma, \mu, \nu, \lambda]$.

Attached property PartialDerivative to ∂#.

Next, we define a tensor g with the `Metric` property and, similarly, define the inverse metric:

```
g_{\alpha\beta} :: Metric.
g^{\alpha\beta} :: InverseMetric.
```

In this case, it is sufficient to specify the components for the original metric. The components for the inverse metric are computed using the function `complete`:

```
g:={ g_{\theta\theta} = r**2, g_{\varphi\varphi} = r**2 \sin(\theta)**2
↪ }.
complete (g, $g^{alpha\beta}$);
```

$$[g_{\theta\theta} = r^2, g_{\varphi\varphi} = r^2(\sin\theta)^2, g^{\varphi\varphi} = (r^2(\sin\theta)^2)^{-1}, g^{\theta\theta} - r^{-2}].$$

Let us define the Christoffel symbols $\Gamma^{\alpha}_{\mu\nu}$:

```
Gamma:= \Gamma^{\alpha}_{\mu\nu} = 1/2g^{\alpha\beta} (\partial)_{\nu}{
↪ g_\beta/mu}} + \partial_{\mu}{g_{\beta\nu}} - \partial_{\beta}}g_
↪ {\mu\nu}});
```

$$\Gamma^{\alpha}_{\mu\nu} = \frac{1}{2} g^{\alpha\beta}(\partial_{\nu} g_{\beta\mu} + \partial_{\mu} g_{\beta\nu} - \partial_{\beta} g_{\mu\nu}).$$

We expand the Christoffel symbols by using the metric tensor. In this case, expansion is applied only to the right-hand side of the definition (after the equal sign). This order of the expansion is determined by the option `rhsonly=True`. The function `evaluate()` implicitly calls SymPy for operations on components (this is another advantage of using the Python infrastructure). To expand trigonometric relationships, SymPy should be called explicitly:

```
evaluate(Gamma, g, rhsonly=True)
map_sympy (Gamma, "expand_trig");
```

$$\Gamma^{\alpha}_{\mu\nu} = \square^{\alpha}_{\mu\nu} \begin{cases} \square^{\varphi}_{\varphi\theta} = (\tan\theta)^{-1} \\ \square^{\varphi}_{\varphi\theta} = (\tan\theta)^{-1} \\ \square^{\theta}_{\varphi\varphi} = -\sin\theta\cos\theta \end{cases}$$

Similarly, for the Riemann tensor $R^{\alpha}_{\beta\mu\nu}$, we write its definition and evaluate its components:

```
R4:=R^{\rho}_{\sigma\mu\nu} = + \partial_{mu}{\Gamma^{\rho}_{\sigma\nu
↪}} - \partial_{\nu}{\Gamma^{\rho}_{\sigma\mu}} + \Gamma^{rho}_{\
↪beta\mu}\Gamma^{\beta}_{\sigma\nu} - \Gamma^{\rho}_{\beta\nu} \
↪ Gamma^{\beta}_{\sigma\mu};
substitute (R4, Gamma)
evaluate (R4, g, rhsonly=True);
```

$$R^{\rho}_{\sigma\mu\nu} = \partial_{\mu}\Gamma^{\rho}_{\sigma\nu} - \partial_{\nu}\Gamma^{\rho}_{\sigma\mu} + \Gamma^{\rho}_{\beta\mu}\Gamma^{\beta}_{\sigma\nu} - \Gamma^{\rho}_{\beta\nu}\Gamma^{\beta}_{\sigma\mu}$$

$$R^{\rho}_{\sigma\mu\nu} = \square^{\rho}_{\sigma\nu\mu} \begin{cases} \square^{\theta}_{\varphi\varphi\theta} = (\sin\theta)^2 \\ \square^{\varphi}_{\theta\varphi\theta} = -1 \\ \square^{\theta}_{\varphi\theta\varphi} = -(\sin\theta)^2 \\ \square^{\varphi}_{\theta\theta\varphi} = 1. \end{cases}$$

The Ricci tensor $R_{\alpha\beta}$ is computed from the Riemann tensor:

```
R2:= R_{\sigma\nu} = R^{\rho}_{\sigma\rho\nu};
substitute (R2, R4)
evaluate (R2, g, rhsonly=True);
```

$$R_{\sigma\nu} = R^{\rho}_{\sigma\rho\nu},$$

$$R_{\sigma\nu} = \square_{\sigma\nu} \begin{cases} \square_{\varphi\varphi} = (\sin\theta)^2 \\ \square_{\theta\theta} = 1. \end{cases}$$

Finally, we compute the scalar curvature R:

```
R:= R = R_{\sigma\nu}g^{\sigma\nu};
substitute (R, R2)
evaluate (R, g, rhsonly=True);
```

$$R = R_{\sigma\nu} g^{\sigma\nu},$$

$$R = 2r^{-2}.$$

Thus, from the user's perspective, component computations in Cadabra 2.x are represented by the additional property `Coordinate` and several functions with the main one being `evaluate()` (it is this function that computes the components).

At the system level, component computations are implemented in Cadabra 2.x through interaction with a scalar computer algebra system, namely, with SymPy.

6. USE OF GRAPHICS IN CADABRA 2.X

The need for graphics in tensor-oriented computer algebra systems is questionable. In our opinion, it is simply not needed. However, in this case, graphics capabilities is nothing more than an additional (side) effect of implementing Cadabra 2.x in the framework of the Python ecosystem. That is why graph plotting in Cadabra 2.x is the same as in Python.

First, we need to choose a plotting library. The following example uses the popular Matplotlib library

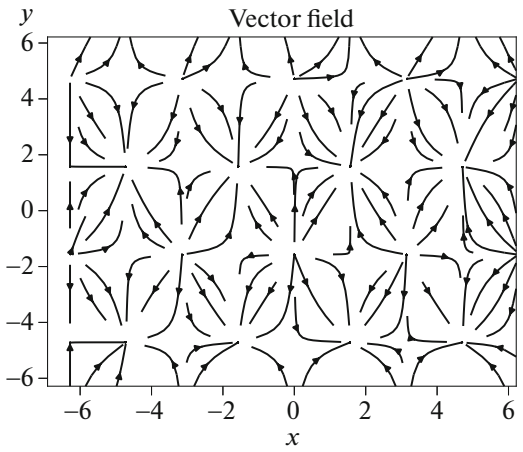

Fig. 1. Image of the vector field (4) constructed using matplotlib.

[23–25]. For numerical computations, the NumPy library is used [26, 27].

Then, we import the modules for `matplotlib` and `numpy`:

```
import matplotlib.pyplot as plt
import numpy as np
```

Let us construct a vector field; i.e., to each point in a space (in our case, on a plane), we assign a vector that originates from this point. Suppose that the vector f has the form

$$f^i(x, y) = \begin{pmatrix} \sin x \cos x \\ \cos y \end{pmatrix}. \quad (4)$$

Let us define a square grid on which the vector field is evaluated. Since grid pitch along the axes is the same, we use only the values for x:

```
x = np.arange(-2*np.pi, 2*np.pi, 0.1)
u = np.sin(x)*np.cos(x)
v = np.cos(x)
uu, vv = np.meshgrid(u,v)
```

The function `meshgrid` from the `numpy` package generates a rectangular grid based on two arrays (in our case, u and v).

Now, we construct the vector field by using the function `streamplot` from the `matplotlib` package:

```
fig = plt.figure()
plt.streamplot(x,x,uu,vv,color= 'black')
plt.title('Vector field')
plt.xlabel('x')
plt.ylabel('y')
```

The resulting plot can be both saved and displayed:

```
fig.savefig("plot.pdf")
display(fig)
```

The image of the vector field is shown in Fig. 1.

We hope that, in the future, Cadabra will include more useful applications than the one illustrated by this simple example.

7. CONCLUSIONS

We can make the following conclusions. From the user's perspective, the main breakthrough of Cadabra 2.x is the implementation of component computations, which allows the system to cover the whole range of necessary tensor operations. From the developer's perspective, the main innovation is rewriting the system by using the Python language and its entire ecosystem. We hope that this will increase interest in Cadabra 2.x when solving problems that involve tensor operations.

ACKNOWLEDGMENTS

This work was supported by the "RUDN University Program 5-100" and the Russian Foundation for Basic Research, project nos. 16-07-00556, 18-07-00567, and 18-51-18005.

REFERENCES

1. MacCallum, M.A.H., Computer algebra in gravity research, *Living Rev. Relativity,* 2018, vol. 21, no. 1, pp. 1–93.
2. Ilyn, V. and Kryukov, A., ATENSOR – REDUCE program for tensor simplification, *Comput. Phys. Commun.,* 1996, vol. 96, no. 1, pp. 36–52.
3. Gomez-Lobo, A.G.P. and Martin-Garcia, J.M., Spinors: A Mathematica package for doing spinor calculus in general relativity, *Comput. Phys. Commun.,* 2012, vol. 183, no. 10, pp. 2214–2225.
4. MacCallum, M., Computer algebra in general relativity, *Int. J. Mod. Phys. A,* 2002, vol. 17, no. 20, pp. 2707–2710.
5. Bolotin, D.A. and Poslavsky, S.V., Introduction to Redberry: The computer algebra system designed for tensor manipulation, 2015, pp. 1–27.
6. Poslavsky, S. and Bolotin, D., Redberry: A computer algebra system designed for tensor manipulation, *J. Phys.: Conf. Ser.,* 2015, vol. 608, no. 1, p. 012060.
7. Fliegner, D., Retery, A., and Vermaseren, J.A.M., Parallelizing the symbolic manipulation program FORM. Part I: Workstation clusters and message passing, 2000.
8. Heck, A., *FORM for Pedestrians,* 2000.
9. Tung, M.M., FORM matters: Fast symbolic computation under UNIX, *Comput. Math. Appl.,* 2005, vol. 49, nos. 7–8, pp. 1127–1137.
10. Peeters, K., Introducing Cadabra: A symbolic computer algebra system for field theory problems, 2007.
11. Peeters, K., Cadabra: A field-theory motivated symbolic computer algebra system, *Comput. Phys. Commun.,* 2007, vol. 176, no. 8, pp. 550–558.
12. Brewin, L., A brief introduction to Cadabra: A tool for tensor computations in general relativity, *Comput. Phys. Commun.,* 2010, vol. 181, no. 3, pp. 489–498.

13. Sevastianov, L.A., Kulyabov, D.S., and Kokotchikova, M.G., An application of computer algebra system Cadabra to scientific problems of physics, *Phys. Part. Nucl. Lett.,* 2009, vol. 6, no. 7, pp. 530–534.
14. Korol'kova, A.V., Kulyabov, D.S., and Sevast'yanov, L.A., Tensor computations in computer algebra systems, *Program. Comput. Software,* 2013, vol. 39, no. 3, pp. 135–142.
15. Kulyabov, D.S., Using two types of computer algebra systems to solve Maxwell optics problems, *Program. Comput. Software,* 2016, vol. 42, no. 2, pp. 77–83.
16. Leeuwen, M.A.A. van, Cohen, A.M., and Lisser, B., *LiE: A Package for Lie Group Computations,* Amsterdam: Computer Algebra Nederland, 1992.
17. Oliphant, T.E., Python for scientific computing, *Comput. Sci. Eng.,* 2007, vol. 9, no. 3, pp. 10–20.
18. Landau, L.D. and Lifshits, E.M., *Teoreticheskaya fizika. Tom II: Teoriya polya* (Course of Theoretical Physics. Vol. 2: Field Theory), Moscow: Fizmatlit, 2012, 8th ed.
19. Perez, F. and Granger, B.E., IPython: A system for interactive scientific computing, *Comput. Sci. Eng.,* 2007, vol. 9, no. 3, pp. 21–29.
20. RFC/RFC Editor, Executor: Leonard, S., 2016.
21. Lamy, R., *Instant SymPy Starter,* Packt Publishing, 2013.
22. Misner, C.W., Thorne, K.S., and Wheeler, J.A., *Gravitation,* San Francisco: W.H. Freeman, 1973.
23. Tosi, S., *Matplotlib for Python Developers,* Packt Publishing, 2009.
24. Vaingast, S., *Beginning Python Visualization: Crafting Visual Transformation Scripts,* Springer, 2009.
25. Müller, A.C. and Guido, S., *Introduction to Machine Learning with Python: A Guide for Data Scientists,* O'Reilly Media, 2016.
26. Idris, I., *NumPy Cookbook,* Packt Publishing, 2012.
27. Oliphant, T.E., *Guide to NumPy,* CreateSpace Independent Publishing Platform, 2015, 2nd ed.

Translated by Yu. Kornienko