

# Computer algebra in Julia

Dmitry S. Kulyabov<sup>1,2,\*</sup> and Anna V. Korolkova<sup>1,†</sup>

<sup>1</sup>*Department of Applied Probability and Informatics,  
Peoples' Friendship University of Russia (RUDN University),  
6 Miklukho-Maklaya St, Moscow, 117198, Russian Federation*

<sup>2</sup>*Laboratory of Information Technologies  
Joint Institute for Nuclear Research  
6 Joliot-Curie, Dubna, Moscow region, 141980, Russian Federation*

Recently, the place of the main programming language for scientific and engineering computations has been little by little taken by Julia. Some users want to work completely within the Julia framework as they work within the Python framework. There are libraries for Julia that cover the majority of scientific and engineering computations demands. The aim of this paper is to combine the usage of Julia framework for numerical computations and for symbolic computations in mathematical modeling problems. The main functional domains determining various variants of the application of computer algebra systems are described. In each of these domains, generic representatives of computer algebra systems in Julia are distinguished. The conclusion is that it is possible (and even convenient) to use computer algebra systems within the Julia framework.

Keywords: Julia language, SymPy computer algebra, Wolfram language, Computer Algebra Systems

## I. INTRODUCTION

For mathematical modeling problems, we want to use the Julia programming language [1–4].

Julia is a problem-oriented language for scientific and engineering computations. It has a simple syntax. To facilitate the migration of researchers and engineers from other languages, Julia borrows good language constructs from MATLAB, R, Python, and FORTRAN [5, 6]. To support interactive development and optimization of the execution time, Julia uses JIT compilation. Intrinsically, Julia is a LISP-like language, and for this reason it seamlessly uses functional constructs. Julia supports libraries written other in programming languages.

A reason for choosing Julia is the consistent implementation of the *single language principle*. The point is that the programming languages for scientific and engineering problems developed in two directions—languages for writing high performance programs (FORTRAN, C, C++) and languages for fast prototyping for nonprogrammers (Matlab, R, Python). Julia tries to resolve the *problem of two languages*<sup>1</sup>.

However, computer algebra problems fall out of the main direction of Julia development.

Taking into account different kinds of problems, we distinguish the following domains of using computer algebra systems.

- Computer algebra systems that provide a large range of possibilities and do not require from the user deep knowledge of programming. Examples are SymPy, Maxima, Axiom, and Reduce.
- Computer algebra languages. They can be used to write nontrivial programs for symbolic computations. Examples are Wolfram and R-Lisp.
- Domain-specific computer algebra systems.

In this paper, we consider computer algebra systems implemented in Julia from the viewpoint of these domains of application.

### A. Structure of the paper

In Section II, we consider a computer algebra system for nonprofessional programmers that uses SymPy library. In Section III, a computer algebra system based on the principles of Wolfram is considered. Section IV is devoted to a problem-oriented symbolic computations language designed for modeling of dynamical systems.

## II. SYMPY.LJ: INTERFACE FOR THE COMPUTER ALGEBRA SYSTEM IN PYTHON

One of the basic advantages of Julia (as well as of Python) is the possibility of using third party libraries. For example, Julia is able to use libraries written in C and FORTRAN. Python has a similar capability (which is the basis of its popularity). It would be silly to fail to use these rich opportunities. For calling Python libraries, Julia uses the package `PyCall`.

The package SymPy (<http://sympy.org/>) is the Python library for symbolic computations. Actually, this

\* kulyabov-ds@rudn.ru

† korolkova-av@rudn.ru

<sup>1</sup> The problem of two languages can be described as follows. A program prototype is written in a language that is most convenient for the developer. The priority is the speed of development. For creating the final program, the code is rewritten in a language that can efficiently use the hardware. Here the priority is the execution speed. Julia supports both paradigms. That is, there is no need to use two languages

is one of the most powerful free symbolic computations tools. Naturally, it can be used from Julia.

By way of example, consider how the functionality of SymPy can be used from the Julia environment. Install PyCall, load it, and then import the library SymPy.

```
import Pkg
Pkg.add("PyCall")
using PyCall
sympy = pyimport("sympy")
```

The further operations are similar to the work in SymPy. Define the symbolic variable `x` and take the sine of this variable:

```
x = sympy.Symbol("x")
y = sympy.sin(x)
```

Now, we calculate  $\sin(\pi)$ :

```
y.subs(x, pi)
```

The result is of the type `PyObject`. In order to use this object, it should be reduced to a numeric type e.g., as

```
y.subs(x, pi) |> float
```

Among other things, this example demonstrates the functional nature of Julia. In this expression, the operator `|>` specifies the action of the function on the right (by analogy with the pipeline in Unix). That is, the expressions `f(x)` and `x |> f` are equivalent.

e expressions `f(x)` and `x |> f` are equivalent. However, the notation above looks somewhat cumbersome. A lot of additional code that has no content has to be written. Fortunately, standard Julia capabilities can be used for the language extensions. To make SymPy operations look similarly to the other computations in Julia, the Python library can be called using the package `SymPy.jl` [7]:

```
import Pkg
Pkg.add("SymPy")
using SymPy
```

Then, the example above takes the form

```
x = Sym("x")
y = sin(x)
subs(y, x, pi) |> float
```

For SymPy objects, the type `Sym` is used.

In principle, this could be the end of discussion. If the reader has experience in using SymPy in Python, then he or she will be able to work with `SymPy.jl` in Julia.

Julia was created in such a way as to facilitate for the people who earlier used other programming languages for scientific and engineering computations (e.g., Matlab, R, or FORTRAN) the migration to Julia. For this reason, Julia contains a lot of syntactic sugar; in particular many operations can be performed in different ways. For example, a symbolic variable can be specified using a constructor

```
x = Sym("x")
```

and using a macro

```
@syms x y z
```

Variables can also be defined using a function that simulates the work with SymPy from Python:

```
x, y = symbols("x y", commutative=false)
```

Here, in addition to declaring symbolic variables, their properties are specified.

Note that the mandatory definition of the variable type contradicts the dynamic nature of Julia. However, in this case we use an external library and, therefore, we play by someone else's rules. Here we have an analogy with Python, which, when working with external libraries, uses the conventions of those libraries rather than replaces its syntax by its own.

`SymPy.jl` gives Julia's user the possibility to manipulate symbolic expressions and provides a convenient interface for this purpose. However (taking into account the performance of modern computers) no speed optimization is achieved.

Here are some examples of working with `SymPy.jl`. In the manipulations with algebraic expressions, the operations of factorization (`factor`) and expansion (`expand`) are very useful:

```
x,y = symbols("x, y")
factor(x^2 - 2x + 1)
```

$$(x - 1)^2$$

```
expand((x-1)*(x+1))
```

$$x^2 - 1$$

In symbolic expressions, Julia constructs, e.g., comprehensions<sup>2</sup> may be used:

```
expand(prod([x-i for i in 1:5]))
```

$$x^5 - 15x^4 + 85x^3 - 225x^2 + 274x - 120$$

This is possible because `SymPy.jl` represents symbolic matrices as arrays with the elements of the type `Julia Sym`, e.g., as `Array{Sym}`<sup>3</sup>.

Let us specify the symbolic matrix

```
x, y = symbols("x y")
m = [x 1; 1 x]
```

---

<sup>2</sup> Comprehensions provide a generic efficient way for constructing arrays. Their syntax resembles the notation of set constructors in the Zermelo–Fraenkel axiomatic.

<sup>3</sup> Naturally, this incurs additional overheads.

$$\begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix}$$

Then, we can perform standard operations on matrices, e.g., multiply them:

`m * m`

$$\begin{bmatrix} x^2 + 1 & 2x \\ 2x & x^2 + 1 \end{bmatrix}$$

`m .* m`

$$\begin{bmatrix} x^2 & 1 \\ 1 & x^2 \end{bmatrix}$$

The majority of standard operations on matrices are also extended for working with symbolic values. By the way, this is an example of the implementation of multiple dispatch.

The functions `factor` and `expand` can simplify the given expression. SymPy has a lot of functions for various simplifications. For example, there is the general function `juliasimplify`, which tries to derive the simplest form of an expression using various heuristics for this purpose: `simplify: simplify:`

```
a = (x + x^2)/(x*sin(y)^2 + x*cos(y)^2)
simplify(a)
```

$$x + 1$$

Equations and systems of equations can be solved using the function `solve`. For example, let us solve the equation

$$x^2 + 3x + 2 = 0$$

for  $x$ :

```
x = symbols("x")
solve(x^2 + 3*x + 2, x)
```

$$\begin{bmatrix} -2 \\ -1 \end{bmatrix}$$

For systems of equations, vector notation may be used:

```
x,y = symbols("x y")
eq1 = x + y - 1
eq2 = x - y - 2
solve([eq1, eq2], [x, y])
```

`Dict{Any,Any}` with 2 entries:

```
y => -1/2
x => 3/2
```

Finally, there is differentiation and integration.

The function `diff` is used to compute derivatives of symbolic expressions. One can compute partial and higher order derivatives. For example, define the functions

```
x, y = symbols("x y")
f(x) = exp(-x) * sin(x)
g(x, y) = x^2 + 17*x*y^2
```

and compute the derivative  $\frac{df(x)}{dx}$ :

```
diff(f(x))
```

$$-e^{-x} \sin(x) + e^{-x} \cos(x)$$

Here we omitted the argument with respect to which the differentiation is performed. However, it may be indicated explicitly:

```
diff(f(x), x)
```

We can also compute the higher order derivative  $\frac{d^3 f(x)}{dx^3}$ :

```
diff(f(x), x, 3)
```

$$2(\sin(x) + \cos(x))e^{-x}$$

and compute the partial derivative  $\frac{\partial g(x,y)}{\partial x}$ :

```
diff(g(x,y), x)
```

$$2x + 17y^2$$

Similarly, symbolic integration can be performed. Thus, we can compute  $\int (x^2 + x + 2) dx$ :

```
x,y = symbols("x y")
integrate(x^2 + x + 2)
```

$$\frac{x^3}{3} + \frac{x^2}{2} + 2x$$

or the double integral  $\int dy \int dx xy$ :

```
integrate(x*y, (x, y))
```

$$\frac{y^3}{2}$$

The definite integral  $\int_0^1 x dx$  can also be computed:

```
integrate(x^2, (x, 0, 1))
```

$$\frac{1}{3}$$

Thus, we have a very rich computer algebra system that can be used directly from the Julia environment.

### III. SYMATA: A COMPUTER ALGEBRA LANGUAGE FOR JULIA

Even though SymPy is an excellent computer algebra system, it has a small drawback. This is a ready-to-use system—a “packaged product”. It provides a user layer for symbolic computations but not a developer layer that makes it possible to write code. For development, one can use the language Symata<sup>4</sup> [8] for symbolic computations. Symata is implemented in Julia. The design of Symata is based on Wolfram [9].

To use this system, the package Symata must be installed:

```
import Pkg
Pkg.add("Symata")
```

Symata is loaded in the standard way:

```
using Symata;
```

Pay attention to the following property of Symata. While SymPy.lj was designed with the aim of using it directly and seamlessly from the Julia environment, Symata is a domain-specific language written in Julia. It is fairly different from Julia. For this reason, one has to explicitly switch between these languages. Switching to Julia is made using the function

```
Julia()
```

Using the function

```
isymata()
```

one can switch back to the Symata mode.

Computations in Symata are similar to the work in Mathematica. For example, suppose that we want to find the value of cosine:

```
expr = Cos(pi * x)
```

$$\text{Cos}(\pi x)$$

Specify the value of the variable  $x$ :

```
x = 1/3
```

$$\frac{1}{3}$$

Now, we get the value of the expression `expr`:

```
expr
```

---

<sup>4</sup> In the case of Symata, the developer layer language and user layer language are identical. Hence, Symata is simultaneously a programming language and a computer algebra system.

$$\frac{1}{2}$$

Modify the value of  $x$ :

```
x = 1/6
```

$$\frac{1}{6}$$

Then the value of the expression also changes:

```
expr
```

$$\frac{3^{\frac{1}{2}}}{2}$$

Reset  $x$ . Then, the numeric value of the expression is not calculated:

```
Clear(x)
```

```
expr
```

$$\text{Cos}(\pi x)$$

Actually, a Wolfram-like language is implemented in Julia. There is no need to describe the syntax of Symata in more detail here. However, it makes sense to consider some differences of Symata from Wolfram. The majority of these differences are due to the desire to make the syntax of Symata closer to the syntax of Julia. For example, comments are marked by the symbol `#`, while in Wolfram, the comments look like

```
(* comment *)
```

For specifying lists, square brackets

```
[a,b,c]
```

$$[a, b, c]$$

rather than curly brackets `{ }` are used. The list elements may be separated by comas or by line break:

```
[
  a
  c + d
  Expand((x+y)^2)
]
```

$$[a, c + d, x^2 + 2 x y + y^2]$$

As in Julia, the argument of a function is specified in parenthesis rather than in square brackets as in Wolfram:

```
f(x)
```

$$f(x)$$

Some functions have infix notation as in Julia.

For example, the function `Map(f,list)` in infix form is written as

```
f % list
```

and the `Apply(x,y)` as

```
x.% y
```

For details, see the Symata documentation.

We may conclude that, on the basis of Julia (using its powerful capabilities in creating domain-specific languages), an extremely convenient language for computer algebra is implemented. It is of interest to mention that Symata uses the library SymPy for many analytical computations.

#### IV. COMPUTER ALGEBRA IN JULIA FOR MATHEMATICAL PROGRAMMING PROBLEMS

In the preceding sections, we considered the implementation of general-purpose computer algebra systems in Julia. However, of great interest are specialized domain-specific applications of computer algebra.

The package `ModelingToolkit.jl` [10] provides a specialized computer algebra language for mathematical modeling problems. For this purpose, it uses the metaprogramming capabilities [11] of Julia. Note that the main mode of operation of `ModelingToolkit.jl` is the batch mode rather than the interactive one.

Symbolic variables are declared by the macro `@variables`:

```
@variables x y
```

Then, these variables can be used in symbolic expressions (in the Julia syntax):

```
z = x^2 + y
```

For modeling continuous dynamical systems, we need derivatives. In `ModelingToolkit.jl`, differential operators are constructed using the macro `@derivatives`:

```
@variables t
@derivatives D'~t
```

Here, the differential operator  $D = \frac{d}{dt}$  is specified. The number of primes `'` indicates the order of the differential operator. We can write the expression

```
z = t + t^2
D(z)
```

At this point, the differential operator does not compute anything because it is a lazy operator. However, we also can obtain the result immediately by using the function `expand_derivatives`:

```
expand_derivatives(D(z))
```

$$1 + 2t$$

Since functions in Julia are first-order objects, the declared symbolic variables are actually functions. When declaring variables, dependences may be indicated explicitly:

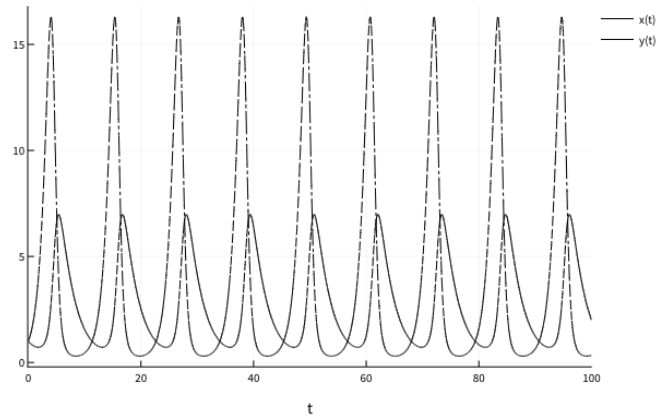


Figure 1. Solution of system (1)

```
@variables t x(t) y(t)
```

This dependence is taken into account in differentiation:

```
z = x + y*t
expand_derivatives(D(z))
```

The last expression is expanded as

```
derivative(x(t), t) + y(t) + derivative(y(t), t) * t
```

By way of example, solve the classical predator-prey problem [12, 13]:

$$\begin{cases} \frac{dx}{dt} = ax - byx, \\ \frac{dy}{dt} = cxy - dy, \end{cases} \quad (1)$$

where  $x$  is the population of prey,  $y$  is the population of predators,  $t$  is time, and  $a$ ,  $b$ ,  $c$ , and  $d$  are the coefficients describing the interaction between species.

First, load the required packages (if needed):

```
import Pkg
Pkg.add("ModelingToolkit")
Pkg.add("OrdinaryDiffEq")
```

Then, we load the package for plot construction:

```
Pkg.add("Plots")
```

and then link the required packages:

```
using ModelingToolkit
using OrdinaryDiffEq
using Plots
```

Now, we are ready to solve the problem. Declare variables and differentiation operators. A part of the variables is declared as parameters.

```
@parameters t a b c d
@variables x(t) y(t)
@derivatives D'~t
```

Now, we simply rewrite system (1) using the syntax of Julia:

```
eqs = [D(x) ~ a *x - b*x*y,
       D(y) ~ c*x*y - d*y]
```

By symbolic manipulations, we reduce the system to the form that is required for the package `OrdinaryDiffEq.jl` [14]:

```
sys = ODESystem(eqs)
```

The further manipulations are performed within the package `OrdinaryDiffEq.jl`. Set the initial values of the variables:

```
u0 = [x => 1.0
      y => 1.0]
```

We also specify the problem parameters:

```
p = [a => 1.1
     b => 0.4
     c => 0.1
     d => 0.4]
```

We will solve the problem on the time interval

```
tspan = (0.0, 100.0)
```

Create the structure describing the entire problem:

```
prob = ODEProblem(sys, u0, tspan, p; jac=true)
```

The additional parameter `jac=true` instructs the system to symbolically generate the optimized Jacobi function for improving the work of differential equation solvers.

Finally, we apply the solver available in the package `OrdinaryDiffEq.jl`:

```
sol = solve(prob)
```

Now, we plot the solution (see Fig. 1):

```
plot(sol)
```

This approach fits well into the idea of a specialized language for scientific and engineering computations. Under this approach, symbolic computations are performed not by using a special universal language, but a special problem-oriented symbolic computation language is created for each direction. This language should seamlessly fit with the basic language (Julia in the case under consideration).

## V. CONCLUSION

The (incomplete) survey of computer algebra systems that can be used with Julia showed that the language itself and its infrastructure are fairly mature. The computer algebra systems in Julia have a wide of applications—systems for nonprofessional programmers, (`SymPy.jl`), powerful computer algebra languages (`Symata.jl`), and domain-specific computer algebra languages (`ModelingToolkit.jl`). This gives hope that the popularity of Julia will grow not only in the domain of numerical computations but also in the domain of symbolic computations.

## ACKNOWLEDGMENTS

The publication has been prepared with the support of the “RUDN University Program 5-100” and funded by Russian Foundation for Basic Research (RFBR) according to the research project No 19-01-00645.

- 
- [1] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah, Julia: A fresh approach to numerical computing, *SIAM Review* 59 (1) (2017) 65–98. [arXiv:1411.1607](https://arxiv.org/abs/1411.1607), doi:10.1137/141000671. URL <https://doi.org/10.1137/141000671https://epubs.siam.org/doi/10.1137/141000671>
  - [2] A. Joshi, R. Lakhanpal, *Learning Julia*, Packt Publishing, 2017.
  - [3] B. A. Tate, F. Daoud, I. Dees, J. Moffitt, *Seven More Languages in Seven Weeks*, 2015.
  - [4] J. Bezanson, S. Karpinski, V. B. Shah, A. Edelman, Julia: A Fast Dynamic Language for Technical Computing (2012) 1–27 [arXiv:1209.5145](https://arxiv.org/abs/1209.5145). URL <http://arxiv.org/abs/1209.5145>
  - [5] J. Bezanson, J. Chen, B. Chung, S. Karpinski, V. B. Shah, J. Vitek, L. Zoubritzky, Julia: dynamism and performance reconciled by design, *Proceedings of the ACM on Programming Languages* 2 (OOPSLA) (2018) 1–23. doi:10.1145/3276490. URL <https://doi.org/10.1145/3276490https://dl.acm.org/doi/10.1145/3276490>
  - [6] S. B. Aruoba, J. Fernández-Villaverde, A Comparison of Programming Languages in Economics, Tech. rep., National Bureau of Economic Research, Cambridge, MA (6 2014). doi:10.3386/w20263. URL <http://www.nber.org/papers/w20263.pdf>
  - [7] Julia interface to SymPy via PyCall. URL <https://github.com/JuliaPy/SymPy.jl>
  - [8] `Symata.jl`. Symbolic mathematics language. URL <https://github.com/jlapeyre/Symata.jl>
  - [9] S. Wolfram, *An Elementary Introduction to the Wolfram Language*, 2015. URL <http://www.wolfram.com/language/elementary-introduction/>
  - [10] `ModelingToolkit.jl`. URL <https://github.com/SciML/ModelingToolkit.jl>
  - [11] R. Lämmel, *Software Languages. Syntax, Semantics, and Metaprogramming*, Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-90800-7. URL <http://link.springer.com/10.1007/978-3-319-90800-7>

- [12] A. J. Lotka, Elements of Physical Biology, Williams and Wilkins Company, Baltimore, 1925.  
URL <https://archive.org/details/elementsofphysic017171mbp>
- [13] D. S. Kulyabov, A. V. Korolkova, L. A. Sevastianov, Two Formalisms of Stochastization of One-Step Models, Physics of Atomic Nuclei 81 (6) (2018) 916–922.
- arXiv:1908.04294, doi:10.1134/S1063778818060248.  
URL <http://link.springer.com/10.1134/S1063778818060248>
- [14] C. Rackauckas, Q. Nie, DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia, Journal of Open Research Software 5 (May) (2017). doi:10.5334/jors.151.

# Компьютерная алгебра на Julia

Д. С. Кулябов<sup>1,2,\*</sup> и А. В. Королькова<sup>1,†</sup>

<sup>1</sup>Кафедра прикладной информатики и теории вероятностей,  
Российский университет дружбы народов,  
117198, Москва, ул. Миклухо-Маклая, д. 6

<sup>2</sup>Лаборатория информационных технологий,  
Объединённый институт ядерных исследований,  
ул. Жолио-Кюри 6, Дубна, Московская область, Россия, 141980

В последнее время на место основного языка научных и инженерных расчётов выдвигается язык Julia. У ряда пользователей возникает желание работать полностью внутри «экосистемы» Julia, подобно тому, как происходит работа в «экосистеме» Python. Для Julia существуют библиотеки, покрывающие большинство потребностей научно-инженерных расчётов. Перед авторами возникла необходимость использовать символьные вычисления для задач математического моделирования. Поскольку основным языком реализации численных алгоритмов мы выбрали язык Julia, то и задачи компьютерной алгебры хотелось бы решать на этом же языке. Авторы выделили основные функциональные области, задающие разные варианты применения систем компьютерной алгебры. В каждой из областей нами выделены наиболее характерные представители систем компьютерной алгебры на Julia. В результате авторы делают вывод, что в рамках «экосистемы» Julia возможно (и даже удобно) использовать системы компьютерной алгебры.

Ключевые слова: Julia, SymPy, язык Wolfram, системы компьютерной алгебра

## I. ВВЕДЕНИЕ

Для задач математического моделирования мы ориентированы на использование языка Julia [1–4].

Язык Julia является проблемно-ориентированным языком для научных и инженерных расчётов. Язык имеет простой синтаксис. Для облегчения перехода научно-инженерных работников, Julia использует удачные языковые конструкции, заимствованные из языков MATLAB, R, Python, FORTRAN [5, 6]. Для поддержки интерактивной разработки и оптимизации времени исполнения Julia использует JIT-компиляцию. Внутри Julia является LISP-подобным языком, поэтому она органично использует функциональные конструкции. Julia поддерживает библиотеки, написанные на других языках программирования.

Одной из причин выбора Julia является последовательная реализация *принципа одного языка*. Дело в том, что в своём развитии языки программирования для научных и инженерных задач разошлись по двум направлениям: языки для создания высокопроизводительных программ (FORTRAN, C, C++) и языки для быстрого прототипирования и использования непрограммистами (Matlab, R, Python). Julia старается решить *проблему двух языков*<sup>1</sup>. Однако задачи компью-

терной алгебры выпадают из основного направления развития языка Julia.

С учётом различно характера возникающих при этом задач, выделим следующие области применения систем компьютерной алгебры:

- Пользовательские системы компьютерной алгебры. Данные системы предлагают пользователю широкий спектр возможностей, не требуя от него глубоких знаний в программировании. Примеры: SymPy, Maxima, Axiom, Reduce.
- Языки компьютерной алгебры. С помощью них можно писать нетривиальные программы символьных вычислений. Примеры: Wolfram, R-Lisp.
- Предметно-ориентированные системы компьютерной алгебры.

В работе рассмотрены системы компьютерной алгебры, реализованные на языке Julia, с точки зрения этих областей применимости.

## A. Структура статьи

В разделе II рассматривается пользовательскую систему компьютерной алгебры, использующей библиотеку SymPy. В разделе III рассматривается язык компьютерной алгебры, реализованный на основе идеологии

числительной техники. Приоритет — скорость выполнения. Язык Julia поддерживает обе эти парадигмы. То есть для него нет необходимости использовать два разных языка программирования.

\* kulyabov-ds@rudn.ru

† korolkova-av@rudn.ru

<sup>1</sup> Проблему двух языков можно представить следующим образом. Прототип программного комплекса пишется на языке программирования, дружественном к разработчику. Приоритет — скорость написания программного кода. Для создания финального программного комплекса код переписывается на языке, позволяющем эффективно использовать ресурсы вы-



языка Wolfram. В разделе IV рассмотрен проблемно-ориентированный язык символьных вычислений для задач непрерывного моделирования динамических систем.

## II. SYMPY.LJ — ИНТЕРФЕЙС К СИСТЕМЕ КОМПЬЮТЕРНОЙ АЛГЕБРЫ НА ЯЗЫКЕ PYTHON

Одной из ключевых основ идеологии языка Julia (как и у языка Python) является возможность использования сторонних библиотек. Например, Julia может подключать библиотеки, написанные на языках C, FORTRAN. Сходной идеологией обладает и язык Python (что и послужило основой его популярности). Поэтому было бы непредусмотрительно не использовать это богатство. Julia использует для вызова библиотек языка Python пакет PyCall.

Пакет SymPy (<http://sympy.org/>) является библиотекой Python для символьной математики. Фактически, это один из наиболее мощных свободных пакетов символьных вычислений. Естественно, что с ним можно работать из Julia.

Рассмотрим для примера, как мы можем использовать функционал SymPy из среды Julia. Установим пакет PyCall, загрузим его, а затем импортируем библиотеку SymPy.

```
import Pkg
Pkg.add("PyCall")
using PyCall
sympy = pyimport("sympy")
```

Дальнейшие операции похожи на работу в SymPy. Определим символьную переменную  $x$  и возьмём синус от этой переменной.

```
x = sympy.Symbol("x")
y = sympy.sin(x)
```

Вычислим теперь  $\sin(\pi)$ :

```
y.subs(x, pi)
```

Результат будет иметь тип `PyObject`. Чтобы его использовать в дальнейшем, следует привести его к какому-либо числовому типу. Например, следующим образом:

```
y.subs(x, pi) |> float
```

Это пример, кроме всего прочего, демонстрирует функциональную природу Julia. В данном выражении оператор `|>` задаёт действие функции справа (по аналогии с конвейером в оболочке Unix). То есть записи  $f(x)$  и  $x |> f$  эквивалентны.

Однако, вышеприведённая запись выглядит несколько тяжеловесной. Для простейших операций требуется писать много дополнительного кода, не несущего никакого содержания. К счастью, можно использовать стандартные возможности Julia для расширений

языка. Для того, чтобы операции SymPy не выделялись на фоне остальных вычислений Julia, вызов питоновской библиотеки производится с помощью пакета `SymPy.lj` [7]:

```
import Pkg
Pkg.add("SymPy")
using SymPy
```

Тогда вышеприведённый пример примет следующий вид:

```
x = Sym("x")
y = sin(x)
subs(y, x, pi) |> float
```

Для объектов SymPy используется тип `Sym`.

В принципе, на этом можно и завершить. Если читатель имел опыт работы с SymPy в Python, он сможет работать и с SymPy.lj в Julia.

Язык Julia создавался таким образом, чтобы на нём легко могли писать программы люди, перешедшие с других языков программирования с научной и инженерной спецификой (таких, как Matlab, R, FORTRAN). По этой причине Julia содержит много синтаксического сахара, в частности одну и ту же операцию можно выполнить несколькими путями. Так, символьную переменную можно задать с помощью конструктора:

```
x = Sym("x")
```

Также можно задать с помощью макроса:

```
@syms x y z
```

Также можно задать с помощью функции, имитирующей работу в SymPy из Python:

```
x, y = symbols("x y", commutative=false)
```

Здесь, кроме всего, кроме объявления символьных переменных указываются и их свойства.

Следует заметить, что обязательное явное задание типа несколько противоречит динамической природе Julia. Однако, в данном случае мы используем внешнюю библиотеку, поэтому играем по чужим правилам. Здесь можно провести аналогию с Python, который, при работе с внешними библиотеками, использует соглашения этой библиотеки, а не подменяет её синтаксис своим.

SymPy.lj даёт пользователю Julia возможность манипулировать символьными выражениями, предлагает для этого удобный интерфейс. Правда (учитывая производительность современных компьютеров) никакой оптимизации по скорости выполнения не производится.

Приведём несколько примеров работы с SymPy.lj.

При операциях с алгебраическими выражениями весьма полезны операции факторизации (`factor`) и расширения (`expand`).

```
x,y = symbols("x, y")
factor(x^2 - 2x + 1)
```

$$(x - 1)^2$$

```
expand((x-1)*(x+1))
```

$$x^2 - 1$$

В символьных выражениях можно применять конструкции Julia, например включения<sup>2</sup>:

```
expand(prod([x-i for i in 1:5]))
```

$$x^5 - 15x^4 + 85x^3 - 225x^2 + 274x - 120$$

Это происходит от того, что SymPy.lj представляет символьные матрицы как массивы с элементами типа Sym, например, как `Array{Sym}`.<sup>3</sup>

Зададим символьную матрицу.

```
x, y = symbols("x y")
m = [x 1; 1 x]
```

$$\begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix}$$

Тогда с матрицами можно производить стандартные операции, например умножить:

```
m * m
```

$$\begin{bmatrix} x^2 + 1 & 2x \\ 2x & x^2 + 1 \end{bmatrix}$$

```
m .* m
```

$$\begin{bmatrix} x^2 & 1 \\ 1 & x^2 \end{bmatrix}$$

Также большинство стандартных операций с матрицами расширены для работы с символьными значениями. Это, кстати, один из примеров реализации множественной диспетчеризации.

Упомянутые выше функции `factor` и `expand` производят некоторое (необходимое нам) упрощение исходного выражения. В SymPy есть много функций для выполнения различного рода упрощений. Например, существует общая функция называемая `simplify`, которая пытается получить простейшую форму выражения, используя для этого разнообразные эвристики: `simplify`:

<sup>2</sup> Включения (comprehensions) задают общий и эффективный способ построения массивов. Их синтаксис похож на обозначения конструктора множеств в аксиоматике Цермело-Френкеля.

<sup>3</sup> Естественно, это влечёт за собой дополнительные накладные расходы.

```
a = (x + x^2)/(x*sin(y)^2 + x*cos(y)^2)
simplify(a)
```

$$x + 1$$

Для решения уравнений и систем уравнений можно использовать функцию `solve`. Например, решим уравнение

$$x^2 + 3x + 2 = 0$$

относительно  $x$ :

```
x = symbols("x")
solve(x^2 + 3*x + 2, x)
```

$$\begin{bmatrix} -2 \\ -1 \end{bmatrix}$$

Для систем уравнений можно использовать векторную нотацию:

```
x,y = symbols("x y")
eq1 = x + y - 1
eq2 = x - y - 2
solve([eq1, eq2], [x, y])
```

`Dict{Any,Any} with 2 entries:`

```
y => -1/2
x => 3/2
```

И наконец, дифференцирование и интегрирование.

Функция `diff` используется для вычисления производной символьных выражений. Можно вычислить частные производные и производные высшего порядка. Зададим, например, функции:

```
x, y = symbols("x y")
f(x) = exp(-x) * sin(x)
g(x, y) = x^2 + 17*x*y^2
```

Вычислим производную  $\frac{df(x)}{dx}$ :

```
diff(f(x))
```

$$-e^{-x} \sin(x) + e^{-x} \cos(x)$$

Мы здесь опустили аргумент, по которому ведётся дифференцирование. Впрочем, его можно и обозначать:

```
diff(f(x), x)
```

Также можем получить производную более высокого порядка  $\frac{d^3f(x)}{dx^3}$ :

```
diff(f(x), x, 3)
```

$$2(\sin(x) + \cos(x))e^{-x}$$

Можно взять и частную производную  $\frac{\partial g(x,y)}{\partial x}$ :

```
diff(g(x,y), x)
```

$$2x + 17y^2$$

Аналогично можно проводить символьное интегрирование. Вычислим  $\int (x^2 + x + 2) dx$ :

```
x,y = symbols("x y")
integrate(x^2 + x + 2)
```

$$\frac{x^3}{3} + \frac{x^2}{2} + 2x$$

Или двойной интеграл  $\int dy \int dx xy$ :

```
integrate(x*y, (x, y))
```

$$\frac{y^3}{2}$$

Можно вычислить и определённый интеграл  $\int_0^1 x dx$ :

```
integrate(x^2, (x, 0, 1))
```

$$\frac{1}{3}$$

Таким образом мы имеем очень богатую систему компьютерной алгебры. И работать с ней можно прямо из среды Julia.

### III. SYMATA — ЯЗЫК КОМПЬЮТЕРНОЙ АЛГЕБРЫ ДЛЯ JULIA

При том, что SymPy является великолепной системой компьютерной алгебры, у неё есть небольшой недостаток. Это готовая система, «работающая из коробки». Она предоставляет «пользовательский слой» для символьных вычислений, но не «слой разработчика», позволяющий писать программный код. Для разработки можно использовать язык программирования для задач символьной математики Symata [8]<sup>4</sup>. Язык Symata реализован на языке Julia. В качестве основы для дизайна языка Symata принят язык Wolfram [9].

Для использования данной системы необходимо установить пакет Symata:

```
import Pkg
Pkg.add("Symata")
```

Загружается Symata стандартным образом:

```
using Symata;
```

Следует обратить внимание на следующее свойство Symata. Если SymPy.lj проектировался таким образом, чтобы его можно было использовать напрямую из среды программирования Julia, бесшовно, то Symata является предметно-ориентированным языком, написанным на Julia. Он достаточно сильно отличается от языка Julia. Поэтому необходимо явно переключаться между этими языками. Переключение в режим языка Julia осуществляется с помощью следующей функции

```
Julia()
```

Обратно в режим Symata можно переключиться с помощью функции

```
isymata()
```

Вычисления в Symata похожи на работу в системе Mathematica. Приведём пример. Пусть нам надо вычислить значение косинуса:

```
expr = Cos(pi * x)
```

$$\text{Cos}(\pi x)$$

Зададим значение переменной  $x$ :

```
x = 1/3
```

$$\frac{1}{3}$$

Теперь собственно можно получить значение выражения `expr`:

```
expr
```

$$\frac{1}{2}$$

Изменим значение переменной  $x$ :

```
x = 1/6
```

$$\frac{1}{6}$$

Тогда изменится и значение вычисляемого выражения:

```
expr
```

$$\frac{3^{\frac{1}{2}}}{2}$$

Сбросим значение переменной  $x$ . Тогда искомое выражение перестанет вычисляться:

```
Clear(x)
expr
```

<sup>4</sup> В случае Symata язык «слоя разработчика» и язык «пользовательского слоя» совпадают. Поэтому Symata является одновременно и языком программирования, и системой компьютерной алгебры.

$\text{Cos}(\pi x)$

Фактически средствами Julia реализован язык Wolfram. Так что, наверное, нет необходимости более подробно освещать здесь синтаксис системы Symata. Впрочем, стоит остановиться на некоторых отличиях языка Symata от языка Wolfram. Большинство этих отличий проистекает из того желания, чтобы Symata наследовала синтаксис Julia. Например, комментарий задаётся символом `#`, в то время как в языке Wolfram комментарий выглядит следующим образом:

```
(* comment *)
```

Для задания списка используются не фигурные скобки `{ }`, а квадратные:

```
[a, b, c]
```

```
[a, b, c]
```

Элементы списка могут разделяться как запятыми, так и переходом на новую строку:

```
[
  a
  c + d
  Expand((x+y)^2)
]
```

```
[a, c + d, x^2 + 2 x y + y^2]
```

Аргумент функции задаётся не в квадратных скобках, как в Wolfram, а в круглых, как в Julia:

```
f(x)
```

```
f(x)
```

Подобно функциям в Julia, некоторые функции приобрели инфиксную нотацию.

Функция `Map(f, list)` приобретает инфиксную форму

```
f % list
```

Функция `Apply(x, y):`

```
x .% y
```

Более подробно можно посмотреть в документации к Symata.

Можно сделать вывод, что на основе Julia (с использованием её мощных возможностей по созданию предметно-ориентированных языков) реализован исключительно удобный язык для компьютерной алгебры. При этом интересно отметить, что для многих аналитических вычислений в языке Symata используется библиотека SymPy.

#### IV. КОМПЬЮТЕРНАЯ АЛГЕБРА В JULIA ДЛЯ ЗАДАЧ МАТЕМАТИЧЕСКОГО МОДЕЛИРОВАНИЯ

В предыдущих разделах мы рассмотрели реализацию систем компьютерной алгебры общего назначения на языке Julia. Но также большой интерес вызывают специализированные, предметно-ориентированные применения компьютерной алгебры.

Пакет `ModelingToolkit.jl` [10] предлагает специализированный язык компьютерной алгебры для задач математического моделирования. Для этого он использует возможности метапрограммирования [11] языка Julia. Отметим, что основным режимом работы для `ModelingToolkit.jl` является не интерактивный режим, а пакетный.

Для задания символьных переменных служит макрос `@variables`:

```
@variables x y
```

После этого данные символьные переменные можно использовать в символьных выражениях (сохраняя синтаксис Julia):

```
z = x^2 + y
```

Для моделирования непрерывных динамических систем необходимо использовать производные. В пакете `ModelingToolkit.jl` дифференциальные операторы строятся с помощью макроса `@derivatives`:

```
@variables t
@derivatives D'~t
```

Здесь задан дифференциальный оператор  $D = \frac{d}{dt}$ . Количество штрихов `'` указывает на порядок дифференциального оператора. Можно записать выражение:

```
z = t + t^2
D(z)
```

Оператор дифференцирования не вычисляет ничего непосредственно в этот момент, поскольку является «ленивым» оператором. Впрочем, мы можем получить результат сразу, применив функцию `expand_derivatives`:

```
expand_derivatives(D(z))
```

```
1 + 2t
```

Поскольку в Julia функции являются объектами первого порядка, то объявленные символьные переменные являются, фактически, функциями. Можно при объявлении переменных явно указать зависимости:

```
@variables t x(t) y(t)
```

Эта зависимость учитывается при дифференцировании:

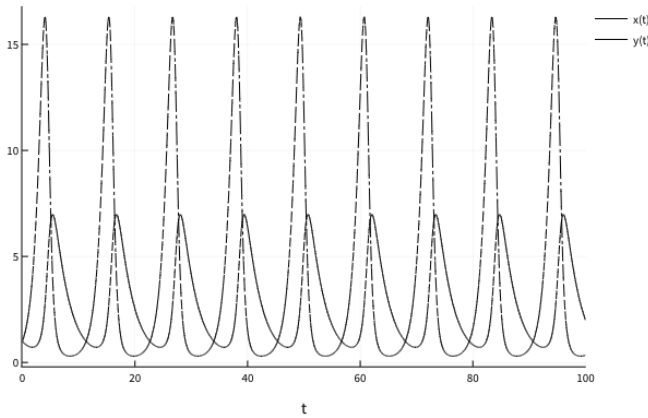


Рис. 1. Решение системы (1)

```
z = x + y*t
expand_derivatives(D(z))
```

Последнее выражение будет раскрыто как

```
derivative(x(t), t) + y(t) + derivative(y(t), t) * t
```

В качестве примера посмотрим, как решается классическая задача типа «хищник–жертва» [12, 13]:

$$\begin{cases} \frac{dx}{dt} = ax - byx, \\ \frac{dy}{dt} = cxy - dy, \end{cases} \quad (1)$$

где  $x$  — количество жертв,  $y$  — количество хищников,  $t$  — время,  $a, b, c, d$  — коэффициенты взаимодействия между видами.

Сначала загрузим необходимые пакеты (при необходимости):

```
import Pkg
Pkg.add("ModelingToolkit")
Pkg.add("OrdinaryDiffEq")
```

Также загрузим пакет для поддержки построения графиков:

```
Pkg.add("Plots")
```

Теперь подключим необходимые пакеты:

```
using ModelingToolkit
using OrdinaryDiffEq
using Plots
```

Собственно, теперь можно решать нашу задачу. Определим переменные и операторы дифференцирования. Часть переменных определим как параметры.

```
@parameters t a b c d
@variables x(t) y(t)
@derivatives D'~t
```

Теперь просто перепишем исследуемую систему (1), используя синтаксис Julia:

```
eqs = [D(x) ~ a * x - b*x*y,
       D(y) ~ c*x*y - d*y]
```

Теперь применим символьные преобразования и приведём нашу систему к виду, необходимому для пакета OrdinaryDiffEq.jl [14]:

```
sys = ODESystem(eqs)
```

Дальнейшие манипуляции проводятся в рамках пакета OrdinaryDiffEq.jl. Зададим начальные значения переменных:

```
u0 = [x => 1.0
      y => 1.0]
```

Также зададим параметры задачи:

```
p = [a => 1.1
     b => 0.4
     c => 0.1
     d => 0.4]
```

Будем решать задачу на следующем временном промежутке:

```
tspan = (0.0, 100.0)
```

Создадим структуру, содержащую всю нашу задачу:

```
prob = ODEProblem(sys, u0, tspan, p; jac=true)
```

Дополнительный параметр `jac=true` указывает системе символически сгенерировать оптимизированную функцию Якоби для улучшения работы решателей дифференциальных уравнений.

И, наконец, применим решатель из пакета OrdinaryDiffEq.jl:

```
sol = solve(prob)
```

Нарисуем график, представленный на рис. 1:

```
plot(sol)
```

Рассмотренный подход очень хорошо вписывается в идеологию специализированного языка для научных и инженерных расчётов. При данном подходе для символьных вычислений используется не специальный универсальный язык, а для каждого направления исследований создаётся свой проблемно-ориентированный язык символьных расчётов. Это язык должен бесшовно стыковаться с базовым языком, в нашем случае, с Julia.

## V. ЗАКЛЮЧЕНИЕ

Наш обзор (далеко не полный) систем компьютерной алгебры для языка Julia показал, что и сам язык, и его инфраструктура являются достаточно зрелыми. При этом системы компьютерной алгебры языка Julia охватывают широкий спектр применения: пользовательские системы компьютерной алгебры (SymPy.jl),

мощные языки компьютерной алгебры (Symata.jl), проблемно-ориентированные языки компьютерной алгебры (ModelingToolkit.jl). Всё это позволяет надеяться, что популярность языка Julia будет возрастать не только в области численных, но и символьных расчё-

тов.

## БЛАГОДАРНОСТИ

Публикация подготовлена при поддержке Программы РУДН «5-100» и при финансовой поддержке РФФИ в рамках научного проекта № 19-01-00645.

- 
- [1] Bezanson J., Edelman A., Karpinski S., Shah V. B. Julia: A fresh approach to numerical computing // *SIAM Review*. — 2017. — 1. — Vol. 59, no. 1. — P. 65–98. — arXiv : 1411.1607.
- [2] Joshi A., Lakhanpal R. *Learning Julia*. — Packt Publishing, 2017. — 316 p. — ISBN: 9781785883279.
- [3] Tate B. A., Daoud F., Dees I., Moffitt J. *Seven More Languages in Seven Weeks*. — 2015. — 318 p. — ISBN: 9781941222157.
- [4] Bezanson J., Karpinski S., Shah V. B., Edelman A. Julia: A Fast Dynamic Language for Technical Computing. — 2012. — P. 1–27. — arXiv : 1209.5145.
- [5] Bezanson J., Chen J., Chung B., Karpinski S., Shah V. B., Vitek J., Zoubritzky L. Julia: dynamism and performance reconciled by design // *Proceedings of the ACM on Programming Languages*. — 2018. — 10. — Vol. 2, no. OOPSLA. — P. 1–23. — Access mode: <https://doi.org/10.1145/3276490><https://dl.acm.org/doi/10.1145/3276490>.
- [6] A Comparison of Programming Languages in Economics : Rep. / National Bureau of Economic Research ; Executor: S. B. Aruoba, J. Fernández-Villaverde. — Cambridge, MA : 2014. — 6. — Access mode: <http://www.nber.org/papers/w20263.pdf>.
- [7] Julia interface to SymPy via PyCall. — Access mode: <https://github.com/JuliaPy/SymPy.jl>.
- [8] Symata.jl. Symbolic mathematics language. — Access mode: <https://github.com/jlapeyre/Symata.jl>.
- [9] Wolfram S. *An Elementary Introduction to the Wolfram Language*. — 2015. — 313 p. — ISBN: 978-1-944183-01-1. — Access mode: <http://www.wolfram.com/language/elementary-introduction/>.
- [10] ModelingToolkit.jl. — Access mode: <https://github.com/SciML/ModelingToolkit.jl>.
- [11] Lämmel R. *Software Languages. Syntax, Semantics, and Metaprogramming*. — Cham : Springer International Publishing, 2018. — XXX, 424 p. — ISBN: 978-3-319-90798-7. — Access mode: <http://link.springer.com/10.1007/978-3-319-90800-7>.
- [12] Lotka A. J. *Elements of Physical Biology*. — Baltimore : Williams and Wilkins Company, 1925. — 435 p. — Access mode: <https://archive.org/details/elementsofphysic017171mbp>.
- [13] Kulyabov D. S., Korolkova A. V., Sevastianov L. A. Two Formalisms of Stochastization of One-Step Models // *Physics of Atomic Nuclei*. — 2018. — Vol. 81, no. 6. — P. 916–922. — arXiv : 1908.04294.
- [14] Rackauckas C., Nie Q. *DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia* // *Journal of Open Research Software*. — 2017. — Vol. 5, no. May.