Discrete & Continuous Models
& Applied Computational Science
ISSN 2658-7149 (online), 2658-4670 (print)

2022, 30 (4) 318–329
http://journals.rudn.ru/miph

Research article

UDC 511.14:004.432
DOI: 10.22363/2658-4670-2022-30-4-318-329

Implementation of hyperbolic complex numbers in Julia language

Anna V. Korolkova[1],
Migran N. Gevorkyan[1], **Dmitry S. Kulyabov**[1, 2]

[1] *Peoples' Friendship University of Russia (RUDN University),*
6, Miklukho-Maklaya St., Moscow, 117198, Russian Federation
[2] *Joint Institute for Nuclear Research,*
6, Joliot-Curie St., Dubna, Moscow Region, 141980, Russian Federation

(received: November 7, 2022; revised: December 12, 2022; accepted: December 19, 2022)

Abstract. Hyperbolic complex numbers are used in the description of hyperbolic spaces. One of the well-known examples of such spaces is the Minkowski space, which plays a leading role in the problems of the special theory of relativity and electrodynamics. However, such numbers are not very common in different programming languages. Of interest is the implementation of hyperbolic complex in scientific programming languages, in particular, in the Julia language. The Julia language is based on the concept of multiple dispatch. This concept is an extension of the concept of polymorphism for object-oriented programming languages. To implement hyperbolic complex numbers, the multiple dispatching approach of the Julia language was used. The result is a library that implements hyperbolic numbers. Based on the results of the study, we can conclude that the concept of multiple dispatching in scientific programming languages is convenient and natural.

Key words and phrases: Julia programming language, multiple dispatch, abstract data types, type conversion, parametric structures, hyperbolic complex numbers

1. Introduction

The Julia programming language [1, 2] is a promising language for scientific computing. At the moment, the Julia language has reached a stable state. By design, Julia solves the *problem of two languages*. This problem lies in the fact that for rapid prototyping, data processing and visualization, an interpreted dynamic language or a mathematical package (Python, Matlab, etc.) is used, and for intensive numerical calculations, the program has to be rewritten in a compiled language with static typing (C/ C++, Fortran).

An illustration of this problem can be seen in Python, which has gained wide popularity as an interface *language-glue*. Numerous wrapper libraries were written on it, which used Python code to call C/C++ and Fortran functions from precompiled libraries. For example, the well-known library

NumPy [3] consists of 51% C code and only 47% Python code (the remaining percentages are divided between C++, Fortran, JavaScript and Unix shell).

The Julia language combines the flexibility of dynamically typed interpreted languages with the performance of statically typed compiled languages.

The basic part of the Julia language is very similar to other scientific programming languages, so it does not cause difficulties in mastering. However, Julia's core is built around the concept of *multiple dispatch* [4], which is rare in other languages. It is in this mechanism that the essential difference of Julia from other languages lies, and its understanding is essential for the full use of all the advantages of Julia.

In the article, the authors paid great attention to illustrating the mechanism of multiple dispatch and other mechanisms that are closely related to it.

In the first part of the article, we give the necessary definitions and illustrate the concept of multiple dispatch with simple examples that allow you to understand the syntax associated with this part of the language and capture the essence of this approach. In the second part, we give an example of the implementation of hyperbolic complex numbers in the Julia language. This example allows you to touch not only multiple dispatch, but also the type casting mechanism, the abstract type hierarchy, overloading arithmetic operators, and specifying user-defined data types.

2. Multiple dispatch

2.1. Common definitions

Dynamic dispatch is a mechanism that allows you to choose which of the many implementations of a polymorphic function (or operator) should be called in a given case [5]. In this case, the choice of one or another implementation is carried out at the stage of program execution. *Multiple dispatch* is based on dynamic dispatch. In this case, the choice of implementation of a polymorphic function is made based on the type, number, and order of the function's arguments. This is how runtime polymorphic dispatch is implemented [6, 7]. Note also that in addition to the term multiple dispatch, the term *multimethod* is also used.

The mechanism of multiple dispatch is similar to the mechanism of overloading functions and operators, implemented, for example, in the C++ language. Function overloading, however, is done exclusively at compile time, while multiple dispatch should work at runtime as well (runtime polymorphism).

2.2. Multiple dispatch in Julia

To illustrate the mechanism of multiple dispatch, we will give the following code example in the Julia language:

```julia
function f(x, y)
  println("Generic implementation")
  return x + y
end

function f(x)
  println("For single argument")
  return x
end

function f(x::Integer, y::Integer)
  println("Implementation for integers")
  return x + y
end

function f(x::String, y::String)
  println("Implementation for strings")
  return x * " " * y
end

function f(x::Tuple{Int, Int}, y::Tuple{Int, Int})
  println("Implementation for tuples of two integer elements")
  return (x[1], x[2], y[1], y[2])
end
```

In this example, we have created five implementations of the *f* function, which differ from each other in different signatures. In terms of the Julia language, this means that one function *f* now has four different *methods.* In the first two methods, we did not use type annotations, so the type of the arguments will be determined either at compile time or at run time (as in interpreted languages). It is also worth noting that Julia uses dynamic JIT compilation (just-in-time), so the compilation stage is not explicitly separated from the execution stage for the user.

The arguments of the following three methods are annotated with types, so they will only be called if the types match the annotations. In the `f` for strings, the `*` concatenation operator is used. The choice of the multiplication sign `*` instead of the more traditional addition sign `+` is justified by the creators of the language by the fact that string concatenation is not a commuting operation, so it is more logical to use the multiplication sign for it, rather than the addition sign, which is often used to denote commuting operations.

The following code snippet illustrates how multiple dispatch works at compile time. The `@show` macro is used to print out the name of the function and the arguments passed to it:

```
@show f(2.0, 1)
@show f(2, 2)
@show f(0x2, 0x1) # numbers in hexadecimal system
@show f("Text", "line")
@show f(3)
@show f([1, 2], [3, 4])
@show f((1, 2), (3, 4))
```

— In the first line, we passed real (floating-point) type arguments to the function, so a generic implementation call was made. Since the operator `+` is defined for floating point numbers, the function succeeded and gave the correct result.

— Methods for integers were called in the second and third lines. Note that the `Integer` type is an *abstract* type and includes signed and unsigned integers from 1 to 16 bytes in size, defined in the language core. Numbers written in hexadecimal are interpreted by default as unsigned integers.

— The method for strings was called on the fourth line. In the fifth line, the method for one argument.
— The sixth line passed two arrays as arguments. The + operation is defined for arrays, so the function ran without error and returned an element-wise sum.
— In the seventh line, the function arguments are tuples consisting of two integers. Since we defined a method for such a combination of arguments, the function worked correctly.

The result of executing the code looks like:

```
Generic implementation
f(2.0, 1) = 3.0
Implementation for integers
f(2, 2) = 4
Implementation for integers
f(0x02, 0x01) = 0x03
Implementation for strings
f("Text", "line") = "Text line"
For single argument
f(3) = 3
Generic implementation
f([1, 2], [3, 4]) = [4, 6]
Implementation for tuples of two integer elements
f((1, 2), (3, 4)) = (1, 2, 3, 4)
```

The above example works correctly in languages that support function overloading and does not demonstrate the specifics of dynamic dispatching, since the types of arguments are known at the compilation stage and are available to the translator.

To test the work of dynamic method calls, consider the following code:

```
  print("Enter an integer:")
  # Read a string and convert to an integer type
  @show n = parse(Int32, readline())
  if n > 0
    x = 1.2; y = 0.1
  else
    x = 1; y = 2
  end
f(x, y)
```

Here, the types of variable values `x` and `y` are not known at compile time, as they depend on what number the user enters during program execution. However, for the case of integer `x` and `y` the corresponding method is called.

3. Hyperbolic numbers

We will use hyperbolic numbers to illustrate the multiple dispatch capabilities of the Julia language, so we will limit ourselves to the definition and basic arithmetic operations.

Hyperbolic numbers [8–11], along with elliptic and parabolic numbers, are a generalization of complex numbers. Hyperbolic numbers can be defined as follows:

$$z = x + \mathrm{j}y, \ \mathrm{j}^2 = 1, \ \mathrm{j} \neq \pm 1. \tag{1}$$

The quantity j will be called the *hyperbolic imaginary unit*, and the quantities x and y will be called the real and imaginary parts, respectively.

For two hyperbolic numbers $z_1 = x_1 + \mathrm{j}y_1$ and $z_2 = x_2 + \mathrm{j}y_2$ the following arithmetic operations are performed.

Addition $z_1 + z_2 = (x_1 + x_2) + \mathrm{j}(y_1 + y_2)$.
Multiplication $z_1 z_2 = (x_1 x_2 + y_1 y_2) + \mathrm{j}(x_1 y_2 + x_2 y_1)$.
Conjugation $z^* = x - \mathrm{j}y$.
Inverse number $z^{-1} = \dfrac{x}{x^2 + y^2} - \mathrm{j}\dfrac{y}{x^2 - y^2}$.
Division $\dfrac{z_1}{z_2} = \dfrac{x_1 x_2 - y_1 y_2}{x_2^2 - y_2^2} + \mathrm{j}\dfrac{x_1 y_1 - x_1 y_2}{x_2^2 - y_2^2}$.

The implementation of hyperbolic numbers is in many respects similar to the implementation of complex ones. Operators `+`, `-`, `*` must be overloaded, and `/`, root extraction, exponentiation, elementary math functions, etc. At the same time, for the purposes of illustrating the mechanism of operation of multiple dispatching, it is arithmetic operations that are of primary interest. This is due to the fact that elementary functions take only one argument, and it is enough to define only one method for them. In the case of arithmetic operators, it is necessary to provide combinations of arguments of different numeric types. So, for example, it should be possible to add a hyperbolic number to an integer, rational, irrational number, which automatically affects not only multiple dispatch, but also type casting mechanisms, an abstract type hierarchy, and default constructor overloading.

Therefore, we will confine ourselves to examples of the implementation of precisely arithmetic operations and that's all, without touching on the more mathematically complex calculations of various elementary functions of a hyperbolic number.

Note that in addition to the term hyperbolic numbers, there are also terms in the literature: double numbers, split complex numbers, perplex numbers, hyperbolic numbers [8, 12–15].

4. Implementation of hyperbolic numbers in Julia

4.1. Declaring a Data Structure

The implementation of hyperbolic numbers in Julia was based on the code for complex numbers available in the official Julia repository. We also used the developments obtained in the implementation of parabolic complex numbers [16]. New type `Hyperbolic` defined with an immutable structure:

```
struct Hyperbolic{T<:Real} <: Number
  "Real part"
  re::T
  "Imaginary part"
  jm::T
end
```

The structure is simple and contains only two fields of parametric type `T`. This requires that the type `T` was a subtype of the abstract type `Real` (syntax `T<:Real`). The type `Hyperbolic` is a subtype of the abstract type `Number` (see figure 1). Thus, hyperbolic numbers are built into an already existing hierarchy of numeric types.

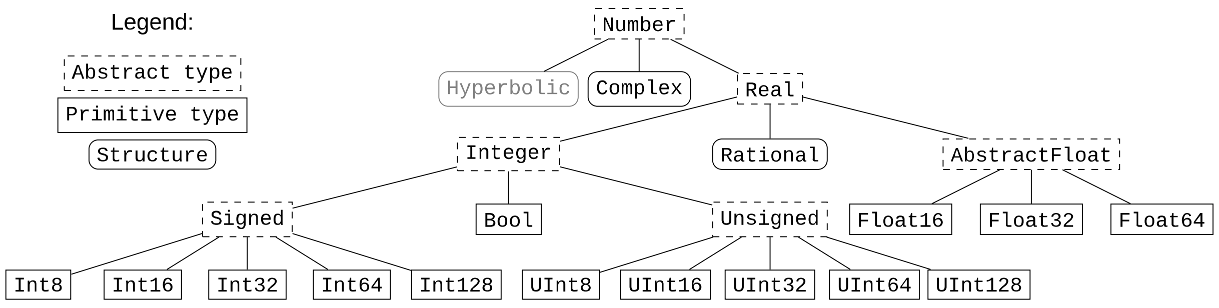

Figure 1. Location of hyperbolic numbers in Julia's type hierarchy

After the structure is defined, a new object of type `Hyperbolic` can be created by calling the default constructor. So, for example, the number $h = 1 + \mathrm{j}3$ is given as follows:

```
h = Hyperbolic{Float64}(1, 3)
```

After creation, you can access the fields of the structure as `h.re` and `h.jm`, but an attempt changing the value of a field of an already existing object will result in an error, since structures are immutable entities:

```
h = Hyperbolic(1, 3).
```

However, if the argument types are different, then the default constructor will not be able to implicitly cast and create a new object. In this case, you must explicitly specify the parametric type

```
# Float64 и Int64
h = Hyperbolic(1.0, 3) # Error
h = Hyperbolic{Float64}(1.0, 3) # Correct
```

4.2. Additional constructors

The default constructor is a normal function whose name is the same as the type name. By creating additional methods for this function, you can create additional constructors to handle various special cases.

So, for example, in order not to specify a parametric type every time, you should add a new constructor of the following form:

```
"""Constructor №2"""
function Hyperbolic(x::Real, y::Real)
  return Hyperbolic(promote(x, y)...)
end
```

The `promote` function casts the arguments passed to it to a common type and returns the result as a tuple. Postfix operator `...` unpacks the tuple and passes its elements as arguments to the constructor function. The language core defines casting rules for all subtypes of the `Real` abstract type, so now the constructor will work correctly for any combination of arguments, as long as the `T<:Real` rule is fulfilled. For example, the following code will work correctly:

```
# Rational и Float64
h = Hyperbolic(1//3, pi)
>> Hyperbolic{Float64}(0.5, 3.141592653589793)
```

We passed a rational number (type `Rational`) and a built-in global constant (number π) of type `Float64` to the constructor. After that, the type casting rule worked and both arguments were cast to the type `Float64` as more general.

Declaring two more additional constructors will allow you to specify hyperbolic numbers with zero imaginary part:

```
"""Constructor №3"""
function Hyperbolic{T}(x::Real) where {T<:Real}
  return Hyperbolic{T}(x, 0)
end
"""Constructor №4"""
function Hyperbolic(x::Real)
  return Hyperbolic(promote(x, 0)...)
end
```

Constructor number 3 is a parametric function that is declared using the `where` construct. The `T` is a subtype of the abstract type `Real`. Constructor number 4 works similarly to constructor number 2.

Two more constructors will allow you to pass other hyperbolic numbers as an argument to the constructor:

```
"""Constructor №5"""
function Hyperbolic{T}(h::Hyperbolic) where {T<:Real}
  Hyperbolic{T}(h.re, h.jm)
end
"""Constructor №6"""
function Hyperbolic(h::Hyperbolic)
  return Hyperbolic(promote(h.re, h.jm)...)
end
```

For more convenience, you can also create a separate constant for the imaginary cost j:

```
const jm = Hyperbolic(0, 1)
```

4.3. Data printing

To be able to print hyperbolic type values in a compact and readable form, you should add the appropriate methods to the `show` function from the `Base` module:

```
function Base.show(io::IO, h::Hyperbolic)
  print(io, h.re, "+", h.jm, "j")
end
```

Function `show` is used when printing data to the console, in particular, it is called by the `println` and macro `@show`. The code and output listings below will assume that the `show` method has been added for hyperbolic numbers.

4.4. Type casting

Before proceeding to the implementation of methods for arithmetic operations with hyperbolic numbers, it is necessary to define the rules for type casting. To do this, create a new method for the function `promote_rule` from the `Base` module:

```
function Base.promote_rule(::Type{Hyperbolic{T}}, ::Type{S})
 ↪  where {T<:Real, S<:Real}
  return Hyperbolic{promote_type(T, S)}
end
function Base.promote_rule(::Type{Hyperbolic{T}},
 ↪  ::Type{Hyperbolic{S}}) where {T<:Real, S<:Real}
  return Hyperbolic{promote_type(T, S)}
end
```

As arguments in `promote_rule` parametric types are specified, which should be cast to one enclosing type. In our case, this is possible if one of the types is a subtype of `Real`, then the enclosing type is `Hyperbolic`.

After adding methods for `promote_rule`, it becomes possible to use functions `promote`, `promote_type` and `convert`:

```
>>h = Hyperbolic(1 // 2)
>>promote(h, 1)
(1//2+0//1j, 1//1+0//1j)
>>promote_type(Hyperbolic{Int64}, Float32)
Hyperbolic{Float32}
```

The first function is already familiar to us. The second allows you to infer the enclosing type not of specific variable values, but of the types themselves. A type in Julia is an object of the first kind (type `DataType`) and can be assigned to other variables, passed as function arguments, and so on.

Function `convert` allows you to convert the type specific value, for example:

```
>>convert(Hyperbolic, 1)
1+0j
```

After adding methods for type casting, you can start adding methods for arithmetic operations. A feature of Julia is the implementation of arithmetic operations not in the form of operators, but in the form of functions. For example, the following calls are correct:

```
>>+(1,2)
3
>>+(1,2,3,4)
10
>>+((i for i in 1:10)...)
55
```

In this regard, adding methods for arithmetic operations is no different from the corresponding process for other functions.

Adding methods for unary operations + and - is carried out as follows:

```
Base.:+(h::Hyperbolic) = Hyperbolic(+h.re, +h.jm)
Base.:-(h::Hyperbolic) = Hyperbolic(-h.re, -h.jm)
```

This is an abbreviated function declaration.

Similarly, methods are added for binary addition, subtraction, multiplication, and division. Here is the code for addition and multiplication:

```
# Binary + and *
function Base.:+(x::Hyperbolic, y::Hyperbolic)
  xx = x.re + y.re
  yy = x.jm + y.jm
  Hyperbolic(xx, yy)
end
function Base.:*(x::Hyperbolic, y::Hyperbolic)
  xx = x.re * y.re + x.jm * y.jm
  yy = x.re * y.jm + x.je * y.re
  return Hyperbolic(xx, yy)
end
```

5. Conclusion

We examined the mechanism of multiple dispatch underlying the Julia language, using the example of the implementation of hyperbolic numbers. This example allowed us to touch upon such concepts of the language as the hierarchy of data types, composite data types, type casting mechanisms, function overloading (creating new methods for functions in terms of the Julia language), etc.

Acknowledgments

This paper has been supported by the RUDN University Strategic Academic Leadership Program.

References

[1] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, “Julia: A fresh approach to numerical computing,” *SIAM Review*, vol. 59, no. 1, pp. 65–98, Jan. 2017. DOI: 10.1137/141000671.

[2] M. N. Gevorkyan, D. S. Kulyabov, and L. A. Sevastyanov, “Review of Julia programming language for scientific computing,” in *The 6th International Conference “Distributed Computing and Grid-technologies in Science and Education”*, 2014, p. 27.

[3] T. E. Oliphant, *Guide to NumPy*, 2nd. CreateSpace Independent Publishing Platform, 2015.

[4] F. Zappa Nardelli, J. Belyakova, A. Pelenitsyn, B. Chung, J. Bezanson, and J. Vitek, “Julia subtyping: a rational reconstruction,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, Oct. 2018. DOI: 10.1145/3276483.

[5] K. Driesen, U. Hölzle, and J. Vitek, “Message dispatch on pipelined processors,” in *ECOOP’95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995* (Lecture Notes in Computer Science), M. Tokoro and R. Pareschi, Eds., Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1995, vol. 952. DOI: 10.1007/3-540-49538-x_13.

[6] R. Muschevici, A. Potanin, E. Tempero, and J. Noble, "Multiple dispatch in practice," in *OOPSLA'08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, ACM Press, Oct. 2008, pp. 563–582. DOI: 10.1145/1449764.1449808.

[7] S. Gowda, Y. Ma, A. Cheli, M. Gwóźzdź, V. B. Shah, A. Edelman, and C. Rackauckas, "High-performance symbolic-numerics via multiple dispatch," *ACM Communications in Computer Algebra*, vol. 55, no. 3, pp. 92–96, Jan. 2022. DOI: 10.1145/3511528.3511535.

[8] I. M. Yaglom, *Complex numbers in Geometry.* Academic Press, 1968, 243 pp.

[9] I. M. Yaglom, B. A. Rozenfel'd, and E. U. Yasinskaya, "Projective metrics," *Russian Mathematical Surveys*, vol. 19, no. 5, pp. 49–107, Oct. 1964. DOI: 10.1070/RM1964v019n05ABEH001159.

[10] D. S. Kulyabov, A. V. Korolkova, and L. A. Sevastianov, *Complex numbers for relativistic operations*, Dec. 2021. DOI: 10.20944/preprints202112.0094.v1.

[11] D. S. Kulyabov, A. V. Korolkova, and M. N. Gevorkyan, "Hyperbolic numbers as Einstein numbers," *Journal of Physics: Conference Series*, vol. 1557, 012027, pp. 012027.1–5, May 2020. DOI: 10.1088/1742-6596/1557/1/012027.

[12] P. Fjelstad, "Extending special relativity via the perplex numbers," *American Journal of Physics*, vol. 54, no. 5, pp. 416–422, May 1986. DOI: 10.1119/1.14605.

[13] W. Band, "Comments on extending relativity via the perplex numbers," *American Journal of Physics*, vol. 56, no. 5, pp. 469–469, May 1988. DOI: 10.1119/1.15582.

[14] J. Rooney, "On the three types of complex number and planar transformations," *Environment and Planning B: Planning and Design*, vol. 5, no. 1, pp. 89–99, 1978. DOI: 10.1068/b050089.

[15] J. Rooney, "Generalised complex numbers in Mechanics," in *Advances on Theory and Practice of Robots and Manipulators*, ser. Mechanisms and Machine Science, M. Ceccarelli and V. A. Glazunov, Eds., vol. 22, Cham: Springer International Publishing, 2014, pp. 55–62. DOI: 10.1007/978-3-319-07058-2_7.

[16] M. N. Gevorkyan, A. V. Korolkova, and D. S. Kulyabov, "Approaches to the implementation of generalized complex numbers in the Julia language," in *Workshop on information technology and scientific computing in the framework of the X International Conference Information and Telecommunication Technologies and Mathematical Modeling of High-Tech Systems (ITTMM-2020)*, ser. CEUR Workshop Proceedings, vol. 2639, Aachen, Apr. 2020, pp. 141–157.

For citation:

A. V. Korolkova, M. N. Gevorkyan, D. S. Kulyabov, Implementation of hyperbolic complex numbers in Julia language, Discrete and Continuous Models and

Applied Computational Science 30 (4) (2022) 318–329. DOI: 10.22363/2658-4670-2022-30-4-318-329.

Information about the authors:

Korolkova, Anna V. — Docent, Candidate of Sciences in Physics and Mathematics, Associate Professor of Department of Applied Probability and Informatics of Peoples' Friendship University of Russia (RUDN University) (e-mail: `korolkova-av@rudn.ru`, phone: +7(495) 952-02-50, ORCID: https://orcid.org/0000-0001-7141-7610)

Gevorkyan, Migran N. — Candidate of Sciences in Physics and Mathematics, Assistant Professor of Department of Applied Probability and Informatics of Peoples' Friendship University of Russia (RUDN University) (e-mail: `gevorkyan-mn@rudn.ru`, phone: +7 (495) 955-09-27, ORCID: https://orcid.org/0000-0002-4834-4895)

Kulyabov, Dmitry S. — Professor, Doctor of Sciences in Physics and Mathematics, Professor at the Department of Applied Probability and Informatics of Peoples' Friendship University of Russia (RUDN University) (e-mail: `kulyabov-ds@rudn.ru`, phone: +7 (495) 952-02-50, ORCID: https://orcid.org/0000-0002-0877-7063)

УДК 511.14:004.432
DOI: 10.22363/2658-4670-2022-30-4-318-329

Реализация гиперболических комплексных чисел на языке Julia

А. В. Королькова[1], М. Н. Геворкян[1], Д. С. Кулябов[1, 2]

[1] *Российский университет дружбы народов,*
ул. Миклухо-Маклая, д. 6, Москва, 117198, Россия
[2] *Объединённый институт ядерных исследований,*
ул. Жолио-Кюри 6, Дубна, Московская область, 141980, Россия

Аннотация. Гиперболические комплексные числа применяются при описании гиперболических пространств. Одним из известных примеров таких пространств является пространство Минковского, играющее ведущее значение в задачах частной теории относительности, электродинамики. Однако такие числа не очень распространены в разных языках программирования. Представляет интерес реализация гиперболических комплексных чисел в языках научного программирования, в частности в языке Julia. В основе языка Julia лежит концепция множественной диспетчеризации (multiple dispatch). Эта концепция является расширением концепции полиморфизма для объектно-ориентированных языков программирования. Разработана библиотека для Julia, реализующая гиперболические комплексные числа. По результатам исследования можно сделать вывод об удобстве и естественности концепции множественной диспетчеризации в языках научного программирования.

Ключевые слова: язык программирования Julia, множественная диспетчеризация, абстрактные типы данных, конвертация типов, параметрические структуры, гиперболические комплексные числа