CEUR-WS.org/Vol-2639/paper-13.pdf

# Approaches to the implementation of generalized complex numbers in the Julia language

Migran N. Gevorkyan[a], Anna V. Korolkova[a] and Dmitry S. Kulyabov[a,b]

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

[b]Laboratory of Information Technologies, Joint Institute for Nuclear Research, 6, Joliot-Curie St., Dubna, Moscow region, 141980, Russia

### Abstract

In problems of mathematical physics in order to study the structures of spaces by using the Cayley–Klein models in theoretical calculations, the generalized complex numbers are essential. In the case of computational experiments, such tasks require their high-quality implementation in a programming language. The proposed small deployment of generalized complex numbers in modern programming languages have several disadvantages. In this article we propose to use the Julia language as the language for generalized complex numbers implemention, not least because it supports the multiple dispatch mechanism.

The paper demonstrates the approach to the implementation of one of the types of generalized complex numbers, namely dual numbers. We place particular emphasis on the description of the use of the multiple dispatch mechanism to introduce a new numerical type. The resulting implementation of dual numbers can be considered as a prototype for a complete software module supporting generalized complex numbers.

### Keywords

complex numbers, parabolic complex numbers, dual numbers, multiple dispatch, Julia

## 1. Introduction

There is the approach that allows one to generalize the complex numbers and get three different classes of generalized complex numbers [1, 2, 3, 4]. In this approach, we set square equation

$$z^2 + pz + q = 0.$$

with a determinant

$$\Delta = p^2 - 4pq.$$

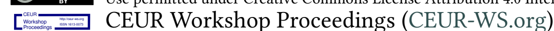
Depending on the sign of the determinant, one gets the following results:

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), Moscow, Russian, April 13-17, 2020

gevorkyan-mn@rudn.ru (M. N. Gevorkyan); korolkova-av@rudn.ru (A. V. Korolkova); kulyabov-ds@rudn.ru (D. S. Kulyabov)

https://yamadharma.github.io/ (D. S. Kulyabov)

0000-0002-4834-4895 (M. N. Gevorkyan); 0000-0001-7141-7610 (A. V. Korolkova); 0000-0002-0877-7063 (D. S. Kulyabov)



© 2020 Copyright for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

- $\Delta < 0$—elliptic (normal complex numbers);
- $\Delta = 0$—parabolic (dual complex numbers);
- $\Delta > 0$—hyperbolic (split complex numbers or double numbers).

Initially, these systems of complex numbers were introduced to describe Cayley–Klein models [5, 6]. However, they have other appliances. In particular, dual complex numbers may be used for problems of automatic differentiation [7].

The goal of this work is to create a pure implementation of dual numbers that is as close as possible to their mathematical definition. We use Julia programming language for this task.

### 1.1. Article structure

The 2 section provides sufficient description of dual complex numbers. We use constructive definition, affecting only operations with dual complex numbers. We try to avoid unnecessary mathematical abstractions. In the section 3 we give a general idea of the multiple dispatch mechanism — the fundamental idea of Julia language. Also, the multiple dispatch is the basis for implementation of any user-defined type in Julia. In the section 4 we describe in detail our implementation of dual complex numbers in Julia.

### 1.2. Notations and conventions

To denote a dual complex unit, we will use the symbol $\varepsilon$.

## 2. Dual numbers

The dual number $z$ is defined algebraically as follows:

$$z = a + \varepsilon b,\ \ a, b \in \mathbb{R},\ \ \varepsilon^2 = 0, \varepsilon \neq 0.$$

The value of $a$ will be called the real part, and $b$ — imaginary. There is no confusion in the terminology, because in this article we not use the ordinary complex numbers.

### 2.1. The algebraic form

Algebraic properties for two numbers $z_1 = a_1 + \varepsilon b_1$ and $z_2 = a_2 + \varepsilon b_2$.

**Addition** $z_1 + z_2 = (a_1 + a_2) + \varepsilon(b_1 + b_2)$.

**Subtraction** $z_1 - z_2 = (a_1 - a_2) + \varepsilon(b_1 - b_2)$.

**Multiplication** $z_1 \cdot z_2 = (a_1 + \varepsilon b_1) \cdot (a_2 + \varepsilon b_2) = a_1 a_2 + \varepsilon(a_1 b_2 + b_1 a_2)$.

**Conjunction** $\bar{z} = a - \varepsilon b$, $z\bar{z} = (a + \varepsilon b) \cdot (a - \varepsilon b) = a^2$.

**Absolute value** $|z| = a$. The modulus of a dual number can be negative, and it can also be calculated by the following formula:

$$|z| = \frac{1}{2}(z + \bar{z}) = \frac{1}{2}(a + \varepsilon b + a - \varepsilon b) = a.$$

**Devision** The division of two dual numbers $z_1$ and $z_2$ is defined for all $z_2$ such that $|z_2| \neq 0$:

$$\frac{z_1}{z_2} = \frac{a_1 + \varepsilon b_1}{a_2 + \varepsilon b_2} = \frac{(a_1 + \varepsilon b_1)(a_2 - \varepsilon b_2)}{(a_2 + \varepsilon b_2)(a_2 - \varepsilon b_2)} = \frac{a_1}{a_2} + \varepsilon \frac{b_1 a_2 - b_2 a_1}{a_2^2}.$$

From the above properties, it can be seen that in imaginary parts of the numbers don't contribute to the real part of the result.

Consider a dual number of the form $c\varepsilon$, where $c \in \mathbb{R}$. This is a dual number with zero absolute value. Such numbers have the following property:

$$(c\varepsilon) \cdot (d\varepsilon) = cd\varepsilon^2 = 0,$$

from which it follows that for any number $c\varepsilon$ there is a number $d\varepsilon$ such that the product of these numbers is 0. Such numbers are called *zero divisors.*

## 2.2. Trigonometric form

A dual number $z$ such that $|z| \neq 0$ can be written as follows:

$$a + \varepsilon b = a\left(1 + \frac{b}{a}\varepsilon\right) = r(1 + \varphi\varepsilon),$$

where the values $r = |z| = a$ and $\varepsilon = Arg z = b/a$ are called *module* and *argument* of the dual number $z$, respectively.

This form of a dual number is some analog of the trigonometric form of an ordinary complex number and we will continue to call it *trigonometric form* of the dual number.

For the conjugate number $\bar{z}$, the trigonometric form is:

$$\bar{z} = r(1 - \varphi\varepsilon),$$

In this case, $|\bar{z}| = |z|$ is an absolute value, and $Arg\bar{z} = -b/a$ is an argument of a dual number in the trigonometric form.

The analogy with the trigonometric form of a complex number continues further when considering multiplication and division. When multiplying two dual numbers $z_1$ and $z_2$, we get

$$z = r_1(1 + \varphi_1\varepsilon) \cdot r_2(1 + \varphi_2\varepsilon) = r_1 r_2\Big(1 + (\varphi_1 + \varphi_2)\varepsilon\Big),$$

in other words, while multiplying, the arguments are added and the modules are multiplied.

In the case of division, we get

$$\frac{z_1}{z_2} = \frac{r_1(1 + \varphi_1\varepsilon_1)}{r_2(1 + \varphi_2\varepsilon)} = \frac{r_1(1 + \varphi_1\varepsilon)r_2(1 - \varphi_2\varepsilon)}{r_2(1 + \varphi_2\varepsilon)r_2(1 - \varphi_2\varepsilon)} = \frac{r_1(1 + (\varphi_1 - \varphi_2)\varepsilon)}{r_2(1 - \varphi_2\varepsilon + \varphi_2\varepsilon)} = \frac{r_1}{r_2}(1 + (\varphi_1 - \varphi_2)\varepsilon),$$

in other words, when dividing, arguments are subtracted, and modules are divided.

Note that in the case of division, the number $z_2$ must have a non-zero module $|z_2| \neq 0$.

In the trigonometric form, the operation of raising to the natural power of $n$ looks especially simple:

$$z^n = \left(r(1+\varphi\varepsilon)\right)^n = \underbrace{r\cdot r\cdot\ldots\cdot r}_{n}\left(1+\underbrace{(\varphi+\varphi+\ldots+\varphi)}_{n}\right) = r^n(1+n\varphi\varepsilon) = a^n + \varepsilon n a^{n-1} b.$$

For finding $\sqrt[n]{z} = \sqrt[n]{r(1+\varphi\varepsilon)}$, $n\in\mathbb{N}$, assume that $\sqrt[n]{z} = z_1$, then by definition $z_1^n = z$, hence

$$r_1^n(1+n\varphi_1\varepsilon) = z = r(1+\varphi\varepsilon),$$

which gives

$$r_1 = \sqrt[n]{r}, \varphi_1 = \frac{\varphi}{n}, \sqrt[n]{z} = \sqrt[n]{r}\left(1+\frac{\varphi}{n}\varepsilon\right).$$

Note that for an odd $n$, the root $\sqrt[n]{z}$ always exists, and for an even $n$ — only for the number $z$ with a non-negative module $|z| = r \neq 0$. In the algebraic form, the formula for the root is:

$$\sqrt[n]{a+b\varepsilon} = \sqrt[n]{a} + \frac{ba^{\frac{1-n}{n}}}{n}\varepsilon.$$

## 2.3. Matrix form

All algebraic operations on dual numbers can be reduced to matrix operations if we consider:

$$\varepsilon \leftrightarrow \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \quad z = a + b\varepsilon \leftrightarrow \begin{pmatrix} a & b \\ 0 & a \end{pmatrix}.$$

Then, for example:

$$z_1\cdot z_2 \leftrightarrow \begin{pmatrix} a_1 & b_1 \\ 0 & a_1 \end{pmatrix}\begin{pmatrix} a_2 & b_2 \\ 0 & a_2 \end{pmatrix} = \begin{pmatrix} a_1a_2 & a_1b_2 + a_2b_1 \\ 0 & a_1a_2 \end{pmatrix} \leftrightarrow a_1a_2 + (a_1b_2 + a_2b_1)\varepsilon.$$

If the programming language supports vectorization, then in theory the matrix form of dual numbers can give you a performance gain. However, the effect is unlikely to be significant in practice.

## 2.4. Taylor series expansion

We can use $\varepsilon^2 = \varepsilon^3 = \ldots = \varepsilon^n = 0\ \forall n\in\mathbb{N}$, to get:

$$\exp(b\varepsilon) = \sum_{n=0}^{\infty}\frac{(b\varepsilon)^n}{n!} = 1 + b\varepsilon + \frac{b^2\varepsilon^2}{2!} + \ldots = 1 + b\varepsilon,$$

$$\exp(a+b\varepsilon) = e^a e^{b\varepsilon} = e^a(1+b\varepsilon).$$

The more general formula is derived from the Taylor series for the function $f(z)$ at the point $a$:

$$f(a+\varepsilon b) = \sum_{n=0}^{\infty}\frac{f^{(n)}(a)(a+\varepsilon b-a)^n}{n!} = \sum_{n=0}^{\infty}\frac{f^{(n)}(a)\varepsilon^n b^n}{n!} = f(a) + f'(a)b\varepsilon + \frac{f''(a)\varepsilon^2 b^2}{2!} + \ldots = f(a) + f'(a)b\varepsilon.$$

As a result, we get the extremely important equation:

$$f(a + \varepsilon b) = f(a) + f'(a)b\varepsilon,$$

which gives a way to calculate the values of functions from a dual number, if the value of the derivative $f'(a)$ is known. On the other hand, the same formula allows one to calculate the value of the first derivative at the point $a$.

### 2.5. Elementary functions of dual numbers

The formula $f(a + \varepsilon b) = f(a) + f'(a)b\varepsilon$ allows you to extend elementary functions to the set of dual numbers, since the right part of the formula contains only the values of the function $f$ from the real number $a$.

Here is a brief summary of basic elementary functions for illustration.

| Trigonometric function | Inverse trigonometric functions |
|---|---|
| $\sin(a + \varepsilon b) = \sin a + b\varepsilon \cos a$ | $\arcsin(a + \varepsilon b) = \arcsin a + b\varepsilon/\sqrt{1 - a^2}$ |
| $\cos(a + \varepsilon b) = \cos a - b\varepsilon \sin a$ | $\arccos(a + \varepsilon b) = \arccos a - b\varepsilon/\sqrt{1 - a^2}$ |
| $\tan(a + \varepsilon b) = \tan a + b\varepsilon/\cos^2 a$ | $\arctan(a + \varepsilon b) = \arctan a + b\varepsilon/(1 + a^2)$ |
| $\cot(a + \varepsilon b) = \cot a - b\varepsilon/\sin^2 a$ | $\mathrm{arccot}(a + \varepsilon b) = \arctan a - b\varepsilon/(1 + a^2)$ |

| Power functions | Logarithmic functions and exponent |
|---|---|
| $(a + \varepsilon b)^n = a^n + na^{n-1}b\varepsilon$ | $\exp(a + \varepsilon b) = \exp\{a\} + b\varepsilon \exp\{a\}$ |
| $\sqrt[n]{a + \varepsilon} = \sqrt[n]{a}\left(1 + \dfrac{b\varepsilon}{na}\right)$ | $\log_c(a + \varepsilon b) = \log_c a + b\varepsilon/a \ln a$ |

### 2.6. Calculation of the first derivative of a real value function by using dual numbers

The formula $f(a+\varepsilon b) = f(a)+f'(a)b\varepsilon$ can also be used to calculate the value of the first derivative of the real function $f(x)$ at the point $a$, if the value of the function $f(a + \varepsilon b)$is known. From the point of view of analytical calculations, this method of finding the derivative does not make sense, since the value $f(a + \varepsilon b)$ is obtained analytically from the same formula. However, it provides a numerical method for obtaining the value of the first derivative using numerical calculations without any additional error. This method of finding the derivative with help of dual numbers is called *automatic differentiation*. It also should mensioned that automatic differentiation using dual numbers is limited to first order derivatives.

Let's first look at a simple example of automatic differentiation, and then discuss the general implementation principle for programming languages.

As an example, let's find the derivative of the function $f(x) = x \sin x$:

$$f(a + \varepsilon b) = (a + \varepsilon b) \sin(a + \varepsilon b).$$

Knowing the value of $\sin(a + \varepsilon b) = \sin a + b\varepsilon \cos a$, it is easy to calculate $f(a + \varepsilon b)$:

$$f(a + \varepsilon b) = (a + \varepsilon)(\sin a + b\varepsilon \cos a) = a \sin a + (\sin a + a \cos a)b\varepsilon.$$

Comparing with the general formula $f(a + \varepsilon b) = f(a) + f'(a)b\varepsilon$, we get the value $f'(a) = \sin a + b\varepsilon \cos a$.

In general, to find $f'(a)$, it is enough to find the value of $f'(a + \varepsilon b)$, take the imaginary part of this dual number and divide it by $b$. In addition, since the choice of the number $b$ is arbitrary, we can choose it equal to 1 and get rid of the need to divide by $b$.

To apply automatic differentiation using dual numbers, we need the programming language which allows user-defined types, as well as overloading arithmetic operations and elementary functions for this type. This is especially easy for object-oriented languages and languages that support function overloading or multiple dispatching.

# 3. Dynamical dispatch in Julia language

The language Julia [8] appeared relatively recently, but has already gained popularity as a language for scientific computing. We will assume that readers are already familiar with this language, and briefly focus only on the concept of *multiple dispatching* [9, 10, 11, 12], which is the basis of the language, its understanding is essential for further presentation.

*Dynamic dispatch* is a mechanism that allows to select the specific implementation of a polymorphic function or operator from a set and call it in a specific case.

*Multiple dispatching* is based on dynamic dispatching. In this case, the choice of exact implementation of polymorphic function is made based on the type, number, and order of the function's arguments. This is the runtime polymorphic dispatch. In addition to the term «multiple dispatching», the term *multimethod* is also used.

The mechanism of multiple dispatch is similar to the mechanism of functions and operators overload, implemented, for example, in the C++ language. Function overloading, however, is performed exclusively at the compilation stage, whereas multiple dispatching must also work at the program run-time (run-time polymorphism).

Julia supports overloading functions at the compilation time if all data types used in the function can be casted at the compilation stage (so called *type stable* functions). The JIT compiler creates efficient implementations for each combination of argument types.

If it is impossible to cast data types at the compilation stage (type unstable function), the dynamic dispatching mechanism is enabled. The compiler will not be able to create a specialized version, but will create a generic version that runs slowly.

This approach allows one to combine the speed of a compiled language with strict static typing with the flexibility of an interpreted language with dynamic typing.

# 4. Dual numbers implementation

## 4.1. Existing implementations of dual numbers in Julia

The authors are familiar at least with two realizations of dual numbers in Julia language:

- type `Dual` in the module for automatic differentiation ForwardDiff [7];
- separate module DualNumbers [13], the development of which is frozen in favor of ForwardDiff.

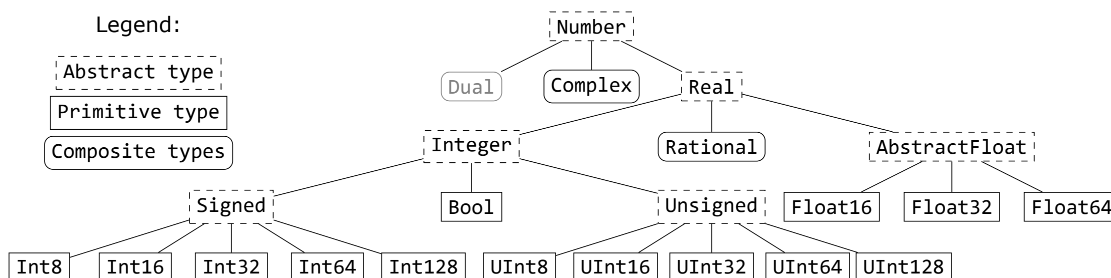



**Figure 1:** Our Dual type in Julia built-in types hierarchy

In this section we will describe our proposed implementation of dual numbers in the Julia language. It is based on a built-in type of complex numbers `Complex` [14], and on the DualNumbers module. The proposed implementation in this paper serves only as a example of creating a custom data type for the Julia language.

We put clarity of presentation at the first place, so many computational optimizations were deliberately omitted in favor of a larger clarity. For example, in the DualNumbers module, the dual number $z = a + \varepsilon b$ can have ordinary complex number components $a$ and $b$. Also in DualNumbers for implementing elementary functions $f(z)$ from dual numbers $z$ a third-party module is used, which defines differentiation rules for functions from the real variable. This allows to automate the definition of the $f(z)$ function by the formula $f(a + \varepsilon b) = f(a) + f'(a)b\varepsilon$, since it is not necessary to explicitly write out derivatives of $f'(a)$.

In our implementation, $a$ and $b$ are only real values and elementary functions are defined explicitly.

### 4.2. Data structure

The adding of any custom data type in Julia starts with the data structure defining. For dual numbers we define the structure `Dual`:

```
struct Dual{T<:Real} <: Number
  "real part"
  x::T
  "imaginary part"
  y::T
end
```

The structure contains only two fields: the real `x` and the imaginary `y` parts of the dual number. Both fields in the structure must have the same parametric type `T`, which must be a subtype of the abstract type **`Real`**, as indicated by the `<:` operator. The `Dual` type itself is a subtype of the abstract type **`Number`**. In this way, the dual numbers are embedded in an existing hierarchy of types (see figure 1).

Immediately after declaring a new structure, we can create variables of the Dual type using the default constructors. We must specify both fields of the structure as arguments of the Dual

function:

```
z = Dual(1, 2) # the same type T of the arguments
Dual{Float64}(1, 2.2) # casting arguments to the same type
```

If the argument type is the same, we don't need to explicitly specify the `T` parameter. If the arguments have different types, the parametric type must be specified. In our example, both arguments are casted to the `Float64` type.

### 4.3. Overloading functions in the show example

To print the values of variables of the Dual type we must to overload the `show`function. This function is intended for formatted printing of data to standard output and to REPL. The `write`, `print`, and `println` functions are used to output a minimal text representation of data. The last two functions print data only to the standard output stream. For simple data, one can make no difference between formatted and compact views and overload only `show` function. Then the other functions will call `show`.

In terms of multiple dispatch, overloading a function means adding a new *method* to the function. To add a method to the `show` function, we define it as follows:

```
Base.show(io::IO, z::Dual) = show(io, z.x, " + ", z.y, "ε")
```

The short syntax for function definition is used here. The prefix `Base.` is needed, since the standard methods of the `show` function are located in the `Base` module. Functions from this module can be called directly without a prefix, but for overloading the `Base` prefix is essential.

It is worth mention that our version of the `show` method is simplified, since it does not take into account special cases, for example, the negative imaginary part.

Next, we will add methods for many functions from `Base`, so that our `Dual` type becomes sufficiently functional.

### 4.4. Additional constructors

Let us create some additional constructors. For this we should overload the default constructor `Dual`.

In total we will set 5 additional constructors:

```
Dual(x::Real, y::Real)= Dual(promote(x, y)...) # 1
Dual{T}(x::Real) where {T<:Real} = Dual{T}(x, 0) # 2
Dual(x::Real) = Dual(promote(x, 0)...) # 3
Dual{T}(z::Dual) where {T<:Real} = Dual{T}(z.x, z.y) # 4
Dual(z::Dual) = Dual(promote(z.x, z.y)...) # 5
```

The first constructor allows us not to specify the `T` parameter every time if the arguments have different types. The `promote` function converts the type of arguments passed to it to a common type and returns the result as a tuple. The 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 abstract type `Real`, so now the constructor will work correctly for any combination of arguments, as long as the `T<:Real` is true. For example, the following code will work correctly:

```
Dual(1//2, π) # 0.5 + π*ε
```

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

The second and third additional constructors allow one to omit the imaginary part if it is equal to zero:

```
Dual{Float32}(1) # 1.0 + 0ε, 2 constructor
Dual(1//2) # 1//2 + 0ε, 3 constructor
```

Constructor number 2 is a parametric function that is declared using the `where` construct. The `T` parameter is a subtype of the abstract type `Real`. Constructor number 3 works similarly to first constructor. The fourth and fifth constructors allow one to pass in an argument to the constructor other dual number.

For more convenience, we can also create a separate constant for the imaginary unit ε:

```
const ε = Dual(0, 1)
```

After overloading arithmetic operations, this constant will allow us to create new dual numbers using an expression as close as possible to their algebraic notation:

```
z = 1 + 2ε
```

## 4.5. Access to structure fields

Structures in Julia are immutable by default, that is, once we create the variable `z`, we can access the fields of the structure itself using the point operator, but we cannot modify the value of these fields:

```
@show z.x z. y # We can read the field value
z.x = 1 # Error! We can't change it.
```

To create mutable data types, the `mutable struct` structure is provided. However, for our example an immutable structure is more appropriate, since the requirement for mutability imposes performance restrictions, which is undesirable for a numeric type.

Julia doesn't have access modifiers for fields like `public` or `private`, so structure fields are always readable. However, it is considered a good programming style to encapsulate structure fields and provide an interface for accessing them. This style is used, for example, for the built-in type of complex numbers.

There are two ways to encapsulate fields. The first method is to create special interface functions for accessing fields. In our case it is sufficient to define the following functions:

```
Base.real(z::Dual) = z.x
Base.imag(z::Dual) = z.y
Base.reim(z::Dual) = (z.x, z.y)
```

The same functions are defined in the built-in Complex module. Next, we should call only these functions everywhere to get the structure fields, instead of accessing the fields by name directly. This will allow developers to refactor the structure in the future, for example, rename or add new fields. Backward compatibility is easy to maintain by rewriting only the interface functions. Performance will not suffer, since the JIT compiler will replace calls to these functions directly with their code, because they are extremely simple (inline functions).

The second approach is to use the `getproperty` function, which has been available since version 0.7 of the Julia language. Overloading this function allows us to set additional names for accessing fields in the structure. So, if we write the following method for example:

```
function Base.getproperty(z::Dual, p::Symbol)
  if p in (:real, :a, :r) # real part aliases
    return getfield(z, :x)
  elseif p in (:imag, :b, :i)  # imaginary part aliases
    return getfield(z, :y)
  else
    return getfield(z, p)
  end
end
```

then we will be able to access the real and imaginary parts of the number $z$ in four different ways:

```
z.x == z.a == z.r == z.real # returns true
z.y == z.b == z.i == z.imag # returns true
```

This approach is more flexible, since when changing the structure for backward compatibility, it is enough to modify only one method `getproperty`, but not all the interface functions (if the structure is complex it can by lots of such functions). In addition, the programmer can use any name from the aliases list for accessing the structure fields.

For the Dual type, we applied both approaches, since the presence of the functions `real` — equivalent to $\mathrm{Re}(z)$ and `imag` — equivalent to $\mathrm{Im}(z)$ is mathematically justified.

### 4.6. Unary functions

The `Base` module defines the functions one and `zero`, which return a multiplication identity element and an addition identity element, respectively. In the case of dual numbers, the multiplication identity is the real unit, and the addition identity is the real zero.

For the Dual type, we define the following single-line methods:

```
Base.one(::Type{Dual{T}}) where T <: Real = Dual(one(T))
Base.one(z::Dual) = Dual(one(z.x))
```

```
Base.zero(::Type{Dual{T}}) where T <: Real = Dual(zero(T))
Base.zero(z::Dual) = Dual(zero(z.x))
```

The first version of the `one` and `zero` functions takes the data type as an argument and returns one or zero of this type, respectively. Since for dual numbers it is 1 and 0 from $\mathbb{R}$, it is sufficient to return a dual number with a zero imaginary part. The real part will be 1 and 0 of the parametric type `T`. For all standard types `T<:Real` the `one` and `zero` methods are defined in the `Base` module, which we used.

The second variant of functions takes a specific object of the `Dual` type as an argument and also returns a identity using methods from the `Base` module.

For complex numbers in `Base`, the conjugation functions `conj`, the absolute value `abs`, and the argument `arg` are defined:

```
Base.conj(z::Dual) = Dual(z.x, -z.y)
Base.abs(z::Dual)  = abs(z.x)
Base.abs2(z::Dual) = z.x^2
function Base.arg(z::Dual)
  @assert !iszero(z.x)
  return z.y / z.x
end
```

For the `arg` method we have provided a check for the inequality of the real part of the dual number to zero, since otherwise we will get a division by zero. Using the standard function `iszero` allows us not to worry about accounting for errors in the representation of real numbers using floating-point numbers. The `@assert` macro throws an exception `AssertionError` if the actual part is equal to zero.

The `inv` function defines the inverse number for this number `z`:

```
function Base.inv(z::Dual)
  @assert !iszero(z.x)
  return Dual(1/z.x, -z.y/z.x^2)
end
```

There should also be an exception for the null real part.

## 4.7. Comparison function

The `Base` module also defines a number of functions that return the truth, if a particular condition is true. Some of these functons are:

- `isreal` is real number;
- `isinteger` is integer;
- `isfinite` the number is finite;
- `isnan` the argument has the type `NaN`;

- `isinf` the argument has the type `Inf`;
- `iszero` the number is an addition identity (zero);
- `isone` the number is a multiplication identity (one).

Methods for these functions for the `Dual` one-to-one case repeat methods for the built-in `Complex` type, so here we do not provide their source code.

In addition to unary functions for dual numbers, it makes sense to implement methods for the comparison operator `==`. Operators in Julia are no different from functions, and adding methods for them is exactly the same. The only difference is that we need to add the character `:` after `Base.`, and since the `==` operator consists of two characters, we must frame it with parentheses:

```
Base.:(==)(z::Dual, u::Dual) = (real(z) == real(u)) && (imag(z) ==
↪  imag(u))
```

It makes sense to compare dual numbers with real numbers as well, if the dual number has a zero imaginary part. In order for the operator to commute for arguments of different types, we must define two methods:

```
Base.:(==)(z::Dual, x::Real) = isreal(z) && real(z) == x
Base.:(==)(x::Real, z::Dual) = isreal(z) && real(z) == x
```

### 4.8. Arithmetic operations

We also need to overload arithmetic operators such as the unary operators `+` and `-` and the binary operators `+`, `-`, `*` and `/`. The implementation of the `+`, `-`, and `*` operators is trivial:

```
Base.:+(z::Dual) = Dual(+z.x, +z.y)
Base.:-(z::Dual) = Dual(-z.x, -z.y)

Base.:+(z::Dual, u::Dual) = Dual(z.x + u.x, z.y + u.y)
Base.:-(z::Dual, u::Dual) = Dual(z.x - u.x, z.y - u.y)

Base.:*(z::Dual, u::Dual) = Dual(z.x * u.x, z.x * u.y + z.y * u.x)
```

In the case of the `/` operator, we should take into account the equality of the divisor to zero:

```
function Base.:/(z::Dual, u::Dual)
  @assert !iszero(u.x)
  return Dual(z.x/u.x, (z.y*u.x-u.y*z.x)/u.x^2)
end
```

For binary operators, we do not need to implement separate cases of arguments of different types, because in this case the promotion mechanism to a common type will call `promote` function automatically.

### 4.9. Types promotion

To make the type promotion mechanism to work, we must define the *type promotion* rules. Without these rules, for example, the next operation fails with an error:

```
>>Dual(1, 2) + 2
ERROR: promotion of types Dual{Int64} and Int64 failed to change any
 ↪ arguments
```

This error occurs because there is no rule that can be used to determine the common type for numbers of the type `Dual{Int64}` and the type `Int64`.

To define such a rule, we have to add a method for the `promote_rule` function from the `Base` module. Any number of the `Real` type can participate in the arithmetic operation with a dual number, since this number can be interpreted as a dual with a zero imaginary part:

```
Base.promote_rule(::Type{Dual{T}}, ::Type{S}) where {T<:Real, S<:Real}
 ↪ = Dual{promote_type(T, S)}
```

We should also provide the rule that will work for operators with two dual numbers with different parametric types `T` and `S`. In this case, we need to find a common type for the `T` and `S` types:

```
Base.promote_rule(::Type{Dual{T}}, ::Type{Dual{S}}) where {T<:Real,
 ↪ S<:Real} = Dual{promote_type(T, S)}
```

### 4.10. Raising to a rational degree

To raise a number to a power, the `^` operator is used, which should also be overloaded for integer and rational powers. In the case of an integer degree, we use the formula

$$(a + \varepsilon b)^n = a^n + nba^{n-1}\varepsilon.$$

There are several special cases to consider:

- if $n = 0$, then $z^n = 1$;
- if $n = 1$, then $z^n = z$;
- if $n > 0$, the formula works for any $a$ and $b$;
- if $n < 0$, the condition $a \neq 0$must be met.

If we consider all these cases, we get the following implementation:

```
function Base.:^(z::Dual, n::Integer)::Dual
  x, y = reim(z)
  if n == 0
    return one(z)
  elseif n == 1
    return z
  elseif n > 1
```

```
      return Dual(x^n, n*y*x^(n-1))
    else #n < 0
      if iszero(x)
        throw(DomainError(:x, "negative exponentiation is only defined
          ↪ for real(z)!= 0"))
      else
        return Dual(x^n, n*y*x^(n-1))
      end
    end
  end
```

If the real part of the dual number is zero (`iszero(x)`) and $n < 0$, the function throws an exception **DomainError** (out of the range of acceptable values).

For a rational degree, the more complex formula is used:

$$(a + \varepsilon b)^{\frac{n}{m}} = a^{\frac{n}{m}}\left(1 + \varepsilon \frac{n}{m}\frac{b}{a}\right) = a^{\frac{n}{m}} + \varepsilon \frac{n}{m} b a^{\frac{n}{m}-1},$$

for which the cases of even and odd $m$ should be provided. For an odd $m$, the range of acceptable values includes $a < 0$, and for an even $m$, you should limit yourself to $a \geqslant 0$:

```
function Base.:^(z::Dual, q::Rational)::Dual
  x, y = reim(z)
  n, d = numerator(q), denominator(q)
  if n == 0
    return one(z)
  elseif isodd(d)
    return Dual(x^q, q*y*x^(q-1))
  else
    if x < 0
      throw(DomainError(:x, "even radical for dual number z is only
        ↪ defined for real(z)>=0"))
    else
      return Dual(x^q, q*y*x^(q-1))
    end
  end
end
```

It makes sense to separately overload the functions for square and cubic roots `sqrt` and `cbrt`:

```
function Base.sqrt(z::Dual)::Dual
  x, y = reim(z)
  if x < 0
    throw(DomainError(:x, "sqrt for dual number z is only defined
      ↪ for real(z)>=0"))
```

```
    else
      return Dual(√x, y/(2*√x))
    end
  end

  Base.cbrt(z::Dual) = Dual(cbrt(z.x), z.y*cbrt(z.x)/(3z.x))
```

### 4.11. Elementary functions

Elementary functions are calculated using the formula $f(a + \varepsilon b) = f(a) + f'(a)b\varepsilon$. Note that many of them are not defined for the case of the zero real part of the number $z = a + \varepsilon b$:

```
function Base.exp(z::Dual)
  e = exp(z.x)
  return Dual(e, e * z.y)
end

Base.log(z::Dual) = Dual(log(z.x), z.y/z.x)
Base.log(b, z::Dual) = Dual(log(b, z.x), (z.y/z.x) * log(z.x))

#======== Trigonometric ========#

Base.sin(z::Dual) = Dual(sin(z.x), +z.y*cos(z.x))
Base.cos(z::Dual) = Dual(cos(z.x), -z.y*sin(z.x))

Base.tan(z::Dual) = Dual(tan(z.x),  z.y/cos(z.x)^2)
Base.cot(z::Dual) = Dual(cot(z.x), -z.y/sin(z.x)^2)

Base.asin(z::Dual) = Dual(asin(z.x),  z.y/sqrt(1-z.x^2))
Base.acos(z::Dual) = Dual(acos(z.x), -z.y/sqrt(1-z.x^2))

Base.atan(z::Dual) = Dual(atan(z.x),  z.y/(1+z.x^2))
Base.acot(z::Dual) = Dual(acot(z.x), -z.y/(1+z.x^2))

#======== Hyperbolic ========#

Base.sinh(z::Dual) = Dual(sinh(z.x), z.y*cosh(z.x))
Base.cosh(z::Dual) = Dual(cosh(z.x), z.y*sinh(z.x))

Base.tanh(z::Dual) = Dual(tanh(z.x), z.y/cosh(z.x)^2)
Base.coth(z::Dual) = Dual(coth(z.x), z.y/sinh(z.x)^2)
```

## 5. Results

The paper describes the preliminary implementation of dual complex numbers and basic operations on them in the Julia language.

## 6. Discussion

The implementation of dual complex numbers on Julia is completely based on the mechanism of multiple dispatching. Thus, we not only implemented a certain set of operations on dual numbers, but also demonstrated the power of this mechanism.

It should also be noted that in contrast to the implementation of the `Dual` type in the automatic differentiation package `ForwardDiff` [7], our proposed implementation is cleaner. For example, in the above package an ordinary complex number can be used as a coefficient before a dual complex unit. It is clear that this is due to the specifics of using dual numbers in this package for automatic differentiation. But this type of number is more likely to belong to quaternions [15, 15, 16], rather than the proper complex numbers.

## 7. Conclusion

In this paper, the prototype of the implementation of generalized complex numbers in the Julia language was made, namely, the implementation of dual complex numbers. Having a multiple dispatching mechanism in the Julia language makes it very easy to implement new numeric types within the existing programming language infrastructure. We propose to further extend this prototype for more general implementation of generalized complex numbers and generalized quaternions.

## Acknowledgments

The publication has been prepared with the support of the Russian Foundation for Basic Research (RFBR) according to the research project No 19-01-00645.

## References

[1] I. M. Yaglom, B. A. Rozenfel'd, E. U. Yasinskaya, Projective Metrics, Russian Mathematical Surveys 19 (1964) 49–107. doi:10.1070/RM1964v019n05ABEH001159.

[2] I. M. Yaglom, Complex Numbers in Geometry, Academic Press, 1968.

[3] B. A. Rozenfel'd, I. M. Yaglom, On the geometries of the simplest algebras, Mat. Sbornik N. S. 28(70) (1951) 205–216.

[4] D. S. Kulyabov, A. V. Korolkova, M. N. Gevorkyan, Hyperbolic numbers as Einstein numbers, Journal of Physics: Conference Series 1557 (2020) 012027.1–5. doi:10.1088/1742-6596/1557/1/012027.

[5] A. Cayley, IV. A sixth memoir upon quantics, Philosophical Transactions of the Royal Society of London 149 (1859) 61–90. doi:10.1098/rstl.1859.0004.

[6] F. Klein, Ueber die sogenannte Nicht-Euklidische Geometrie, in: Gauß und die Anfänge der nicht-euklidischen Geometrie, volume 4 of *Teubner-Archiv zur Mathematik*, Springer-Verlag Wien, Wien, 1985, pp. 224–238. doi:10.1007/978-3-7091-9511-6_5.

[7] Forward Mode Automatic Differentiation for Julia, 2020. URL: https://github.com/JuliaDiff/DualNumbers.jl.

[8] The Julia Language, 2020. URL: https://julialang.org/.

[9] 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 (2018) 1–23. doi:10.1145/3276490.

[10] F. Zappa Nardelli, J. Belyakova, A. Pelenitsyn, B. Chung, J. Bezanson, J. Vitek, Julia subtyping: a rational reconstruction, Proceedings of the ACM on Programming Languages 2 (2018) 1–27. doi:10.1145/3276483.

[11] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah, Julia: A fresh approach to numerical computing, SIAM Review 59 (2017) 65–98. doi:10.1137/141000671. arXiv:1411.1607.

[12] J. Bezanson, S. Karpinski, V. B. Shah, A. Edelman, Julia: A Fast Dynamic Language for Technical Computing (2012) 1–27. arXiv:1209.5145.

[13] Julia package for representing dual numbers and for performing dual algebra, 2020. URL: https://github.com/JuliaDiff/ForwardDiff.jl.

[14] Official Julia language GitHub repository, 2020. URL: https://github.com/JuliaLang/julia.

[15] W. R. Hamilton, Elements of Quaternions, Cambridge University Press, Cambridge, 1866. doi:10.1017/CBO9780511707162.

[16] A. P. Yefremov, Quaternions and Biquaternions: Algebra, Geometry and Physical Theories, 2005. arXiv:0501055.