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

2023, 31 (1) 5–26
http://journals.rudn.ru/miph

Research article

UDC 537.8:512.723
DOI: 10.22363/2658-4670-2023-31-1-5-26
EDN: VNJCSU

Julia language features for processing statistical data

Migran N. Gevorkyan[1], **Anna V. Korolkova**[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: January 16, 2023; revised: March 13, 2023; accepted: April 10, 2023)

Abstract. The Julia programming language is a specialized language for scientific computing. It is relatively new, so most of the libraries for it are in the active development stage. In this article, the authors consider the possibilities of the language in the field of mathematical statistics. Special emphasis is placed on the technical component, in particular, the process of installing and configuring the software environment is described in detail. Since users of the Julia language are often not professional programmers, technical issues in setting up the software environment can cause difficulties that prevent them from quickly mastering the basic features of the language. The article also describes some features of Julia that distinguish it from other popular languages used for scientific computing. The third part of the article provides an overview of the two main libraries for mathematical statistics. The emphasis is again on the technical side in order to give the reader an idea of the general possibilities of the language in the field of mathematical statistics.

Key words and phrases: Julia programming language, statistic processing

1. Introduction

In this paper we give a brief overview of Julia [1] programming language capabilities in the field of mathematical statistics. Julia is a fast compiled language with dynamic typing, originally developed for scientific computing. The language is relatively new, however, it has already reached version 1.8 and the core of the language is quite stable. An impressive number of modules have been created for Julia and several books have been written [2–4].

There are a number of arguments in favor of learning and using the Julia language:

— Just-in-Time compilation (JIT) [5] allows you to simultaneously achieve high performance and ease of use of the interpreted language. Single-threaded programs in Julia have the performance of programs in C/C++ and Fortran [6] and significantly exceed the interpreted languages, such as R, Python, Matlab, SciLab, etc.

— The syntax of Julia is simple and for researchers familiar with Python, Fortran and R languages, it will not be difficult to master it at a basic level in the shortest possible time.
— The language has built-in extensive capabilities for parallel and distributed computing, which are constantly being refined.

The authors tend to give a general idea of Julia language's available capabilities in the field of mathematical statistics and demonstrate a number of examples that allow one to quickly grasp the features of the language and move on to use it. At the beginning of the article we give a step-by-step description of the configuration of the working environment for Unix-type systems (macOS, GNU/Linux) and Windows. We do not give a consistent description of the syntax of the language, but focus on some specific features (dynamic dispatching, custom data types) that distinguish Julia from most popular programming languages.

Libraries for mathematical statistics for Julia are combined under the general name Julia Statistics [7, 8] and a separate section is allocated for them on the official forum of language developers [9]. In the main part of this paper, we give an overview of the modules `StatsBase` and `Distributions` [10, 11], comparing their functionality with the libraries of the `R` language and the `scipy.stats` library of Python [12].

2. Installation and configuration of Julia environment

There are several ways of programs development in Julia languages.
— Using REPL-shell (read-eval-print loop) in interactive mode, by running the `julia` command from the terminal and entering instructions that will be executed immediately, and the user will see the returned result.
— By saving the program source code to files with the extension `jl` and then passing them for compilation and launching to the julia JIT compiler (same `julia` command).
— By using interactive shells, such as Jupiter Notebook [13] and Pluto [14].

We will describe the process of Julia installation, as well as Jupiter interactive shell and all necessary modules in the GNU/Linux and Windows environments. The installation does not require superuser rights and it can be performed remotely by connecting via ssh, which can be convenient if calculations are supposed to be performed on a remote server.

2.1. Installing Julia and the necessary packages

On the official Julia website, in the `downloads` section, binary files for many systems are presented. Download the archive for the 64-bit version of GNU/Linux:

```
wget https://julialang-s3.julialang.org/bin/linux/x64/1.8/juli⌋
↪  a-1.8.5-linux-x86_64.tar.gz
↪  --no-check-certificate
```

Please note that the url may change, as it clearly indicates the current version of the Julia distribution. Extract the files from the downloaded archive:

```
tar -xvzf julia-1.8.5-linux-x86_64.tar.gz
```

The directory `julia-d386e40c17` will be created (or with another alphanumeric combination), which we will rename to just `julia`:

```
mv julia-d386e40c17/ julia
```

```
export PATH="~/julia/bin:$PATH"
```

In the case of the Windows operating system, we will describe the installation of the portable version. In the same section of the official website, download the 64-bit (portable) version for Windows and unpack the archive, for example, into the following directory:

```
E:\Program Files\julia
```

In this directory, we will create the folder `depot`, and in it, the folder `config`, in which we will create an empty text file `startup.jl`, which we will need next. The Julia directory `depot` will host an index of modules from the official repository, as well as installed modules and additional libraries. The location of this directory is non-standard and in order for the Julia JIT compiler to recognize it correctly at startup, you should create an environment variable `JULIA_DEPOT_PATH` and assign it a value:

```
E:\Program Files\julia\depot
```

We also added the path to the Julia JIT compiler to the variable `PATH` (executable file julia.exe)

```
E:\Program Files\julia\bin
```

After the installation is complete, run the Julia command shell. To do this, run the command `julia` in the console. In the case of Windows, it is recommended to use PowerShell or the new Windows Terminal application [15]. After launching, press the key `]` and switch to package management mode, where you run the command `update`, which will download the package index. You can also immediately install the necessary packages using the command `add`, for example

```
add StatsBase Distributions Pluto Plots
```

The built-in package manager saves all package-related files to the storage directory (depot), which is pointed to by the environment variable `JULIA_DEPOT_PATH`. No other directories are involved.

On Unix systems — if the variable `JULIA_DEPOT_PATH` is not defined — the corresponding directory is created in the user directory and is called `.julia`. In it, you should also manually create a directory `config` with the configuration file `startup.jl`.

At this stage, the installation and configuration of the compiler is complete and we will proceed to the installation of additional tools that may be needed to write programs on Julia.

2.2. Jupyter installation

Julia language code can be executed in the Jupyter environment, for which you should install the Jupyter Notebook kernel, which is included in the package `IJulia`. During the installation, the built-in Julia package manager automatically downloads the Python distribution miniconda [16], places it in the storage directory and installs with its help all the necessary python packages, including Jupyter Notebook. Since most Julia users probably already have a Python distribution installed on the system, we will show you how to use the already installed Jupyter in Julia. Even if there is no Python distribution in the system, it seems more practical to install it separately, as this will allow better control of the packages used.

Next, we will describe the process of installing Jupiter using the Miniconda distribution into the user's local directory. Download the installation script from the official website:

```
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Lin⌋
 ↪  ux-x86_64.sh
```

and start the installation process:

```
bash Miniconda3-latest-Linux-x86_64.sh
```

During the installation process, you must read and accept the license agreement by using the key `Enter` to scroll through the text and typing the word `yes` to accept it. After that, the installer will prompt you to select the directory where the distribution directory tree will be copied. By default, this is the directory `miniconda3` in the user's home directory. Let's leave it unchanged, for which you should press `Enter`. The process of downloading the necessary files will begin, after which the installer will offer to add the path to the Python interpreter to the environment variable `PATH`. You should agree by typing the word `yes` and pressing `Enter`. Then check that the following line has been added to the file `.bashrc` located in the home directory:

```
export PATH="~/miniconda3/bin:$PATH"
```

where `~/miniconda3/bin` is the path to the miniconda directory. If the installer did not add this path automatically, you can do it manually.

After the anaconda installation is completed, the command `conda` will be available with which you can manage the installed Python modules. Let's use this command to install the modules we need (Numpy, SciPy, Matplotlib and Jupiter):

```
conda install numpy matplotlib scipy jupyter
```

The process of downloading and unpacking the required files can take considerable time, and after completion, the miniconda directory will occupy about 2.5 GB of disk space. To check the correctness of the installation, run Jupiter by running the following command:

```
jupyter notebook --notebook-dir=~ --port=7000
```

The interactive shell session will start. If launched on a local computer, a browser will automatically open with a list of files and directories of the home directory (option `--notebook-dir=~`). If you run it on a remote computer, you should add the option `--no-browser`, then you can connect to the session that has started remotely by entering the address of the remote computer into the local network in the browser address bar or organize an ssh tunnel.

Now, in the file `startup.jl`, which we previously created, but left empty, we should add local environment variables that will point to the locations of the executable files of the python interpreter and the jupyter shell:

```
ENV["PYTHON"] = "~/miniconda3/bin/python"
ENV["JUPYTER"] = "~/miniconda3/bin/jupyter"
```

Note that the variable `ENV` is a dictionary to which, when the compiler is started, system and user environment variables are added, as well as a number of local Julia parameters. This dictionary is available in any julia program for reading and modification, which we used by adding two new keys to it.

After that, run REPL Julia with the command `julia` and install the necessary packages:

```
add PyCall PyPlot IJulia
```

During the installation process, the package manager will see that the variables `PYTHON` and `JUPITER` have been assigned values and the local copy of miniconda will not be installed.

After the installation is completed, when Jupiter Notebook is launched, the Julia kernel will be available and it will be possible to create and open interactive notebooks with scripts in the Julia language.

In addition to `Julia`, we also installed the package `PyCall`, which greatly simplifies calling functions from python modules, and the package `PyPlot`, which makes it possible to use the library `Matplotlib` in Julia. In this article, we will not use the capabilities of these packages, but for those users who are used to standard Python scientific libraries, they may be useful, since they transfer the usual functionality to Julia.

2.3. Pluto shell as an alternative to Jupiter

Using Jupyter with the Julia language has at first glance an unobvious drawback associated with a fundamental feature of the architecture of the language itself — multiple dispatching of functions. In the case of Jupyter, it manifests itself as follows: when initially creating a function in a separate cell and executing this cell once, no problems arise, however, if the programmer decides to change the body of this function without changing the signature of the arguments, then re-executing the cell with the modified code will lead to an error. If the list of arguments has been modified, there will be no errors during execution, but a new method will be created or, in other terminology, an overloaded version of the function will be created. This difficulty can be overcome by restarting the kernel, but this makes the process uncomfortable if the cells contain resource-intensive calculations, which will have to be done again every time. There will definitely be such cells, since the initial initialization of graphical libraries for data visualization in Julia is extremely slow, and the main advantage of Jupyter is precisely the interactive display of the results of various data visualization.

The Julia development community has created an alternative shell called `Pluto.jl`. At the moment, the repository of this package [17] ranks second in the number of stars on GitHub, second only to the Julia compiler itself [18].

Shell `Pluto.jl` is generally similar to `Jupyter`, but has two key differences:

— reactivity (reactive);
— no hidden states (no hidden workspace state).

Reactivity lies in the fact that all cells of the interactive notebook are immediately restarted if the variables on which they depend are modified, even if these variables are contained in other cells.

The absence of hidden states means that if a cell is deleted, then all the variables, functions and data structures contained in it are deleted from memory and become inaccessible. It is also impossible to redefine variables and functions in neighboring cells, which removes the problem with implicit function overloading.

To install `Pluto`, just run `add Pluto` in package management mode in Julia REPL. No additional manipulations are required, since `Pluto` is written in Julia only. At the time of writing, this package has reached version 0.19.22, but it works quite stably. Among the disadvantages, it can be noted that the interface is too minimalistic, as well as the demands on the amount of RAM.

To run the shell in Julia REPL mode, follow these instructions

```
import Pluto
Pluto.run()
```

At the same time, the browser will immediately be launched with a welcome interface, and a link will be displayed in the console, which can be used for remote connection.

Let's note some features of the interactive notepad. To store the contents of the notebook, a simple text file with the extension `jl` is used, the entire code of the cells is stored as a regular code in the Julia language, and standard code comments are used to store meta information. This makes it possible to execute Pluto notebooks like regular Julia programs, passing them to the JIT compiler for execution.

Pluto has built-in support for local package environments. It automatically detects the packages used by looking at instructions `used` and `import` and downloading the necessary packages to the local directory. This is useful if you need to transfer the created notebooks to third-party users, as well as fix specific versions of libraries. However, this behavior can be disabled if desired, for which the following code should be added to the first cell

```
begin
  using Pkg
  Pkg.activate()
end
```

This command will disable the local package manager and Pluto will use the standard Julia environment that was created when the package update was initially launched.

This code snippet illustrates another feature of Pluto which is called reactivity. By default, each cell can contain only one line of code. In the case of several lines, they must be framed with the construction `begin` ... `end`. This may cause some inconvenience, but the shell itself determines such cells and offers to automatically insert `begin` and `end`.

Pluto notepad allows you to add cells with comments in markdown format in combination with LaTeX formulas. Unlike jupiter notebooks in Pluto, these are not special cells, but standard ones with a multiline string preceded by the `md` modifier, for example:

```
md"""# Header
The text of the comment and the equation $\dot{x} = f(x)$
"""
```

Finally, we note that Pluto includes the module `PlutoUI`, which allows you to create interactive graphical interface elements such as sliders, drop-down lists, text input fields, etc. and bind variables to them. This allows you to add interactivity to the notebooks being created, which is useful, for example, for selecting parameters for certain functions. Due to reactivity, when changing the values of variables, all graphs that depend on them will also be rebuilt.

3. The main features of the Julia language

The Julia language was originally created for the field of scientific programming and its syntax is very similar to the syntax of the Fortran and Python languages, which are well known to specialists in scientific computing. However, at the same time, it contains some specific features, and without knowing them, it will be difficult to use third-party libraries effectively. We will illustrate all the features with examples from probability theory and mathematical statistics.

3.1. Custom data structures

One of the distinctive features of the Julia language is the high performance of data types created by the user himself. In many modules there is large number of custom data types and functions.

A composite data type in the first approximation resembles a structure from the C-language. It is specified using the construct `struct`, inside which the fields of the structure with the type annotation are listed. As an example, consider setting a structure that stores the parameters of a normal distribution.

```
"Normal distributions"
struct Normal
    "first moment"
    μ::Real
    "standard deviation"
    σ::Real
end
```

Let's list some important features.

— Since Julia provides full Unicode encoding support, Greek letters and other symbols that are standard for mathematical formulas can be used as field designations.
— A composite type and its fields can be provided with documentation lines that explain the purpose of the structure and its fields. These strings are similar to Python doc-strings, with the difference that they can be supplied to almost any object and they must be specified before, not after the declaration.
— Julia is a dynamically typed language, but it supports type annotations that can be used by the compiler for code optimization and to limit the types of variables passed to functions when they are called and to structures when they are initialized.

After defining the structure, you can create objects of type `Normal` using the default constructor.

```
N = Normal(0, 1)
@show typeof(N)
@show N.μ, N.σ
```

The macro `@show` prints the line of code that is passed to it and the result of executing this line of code. So, in the example above, the following will be printed to standard output:

```
typeof(N) = Normal
(N.μ, N.σ) = (0, 1)
```

The default constructor is created automatically, but it can be set explicitly in the body of the structure, for example, if you need to limit the scope of acceptable values of the fields of the structure. After the checks, it is necessary to allocate memory for the fields of the structure using a special function `new`.

```
struct Normal
  μ::Real
  σ::Real
  function Normal(μ, σ)
    if σ == 0
      throw(ArgumentError("σ != 0"))
    end
    return new(μ, σ)
  end
end
```

Only one main constructor can be defined in the structure body. If you need to define additional constructors, they should be set outside the structure. So, you can define a constructor without arguments, which will set the parameters of the standard normal distribution.

```
function Normal()
  return Normal(0.0, 1.0)
end
```

3.2. Multiple dispatch

Julia implements a multiple dispatching mechanism [5, 19, 20], which, according to the developers, is a more flexible mechanism compared to the object-oriented approach applied to mathematical applications.

Each function in Julia can have many implementations called *methods*. Implementations have the same name, but differ from each other both in the number of arguments and their types. When calling a function, the compiler analyzes the arguments passed to it and calls the desired implementation. Various operators such as +, - are also functions and can be overridden for any new data type.

To illustrate multiple dispatching, we additionally define a structure that stores the parameters of the exponential distribution:

```
struct Exponential <: Distribution
  λ::Real
  function Exponential(λ)
    if λ <= 0
      throw(ArgumentError("λ > 0"))
    end
    return new(λ)
  end
end
```

Now we implement two functions that calculate the PDF of normal and exponential distributions:

```
function pdf(d::Normal, x)
    return 1/(sqrt(2*π)*d.σ) * exp(-(x-d.μ)^2 / (2*d.σ^2))
end
```

```
function pdf(d::Exponential, x)
    return d.λ * exp(-d.λ*x)
end
```

It should be noted that the first arguments of the function are provided with type annotations. This is done so that the compiler can call the desired implementation depending on the type of the first argument:

```
N = Normal(0, 1)
E = Exponential(2)
@show pdf(N, 3) # <- call of the implementation for the normal
 ↪  distribution
@show pdf(E, 2) # <- call of the implementation for the
 ↪  exponential distribution
```

In addition, Julia allows you to automatically vectorize a scalar function, that is, apply it to each element of some array, without having to implement an additional method. To do this, it is enough to use a special syntax:

```
pdf.(N, [1, 2, 3, 4, 5])
```

To achieve a similar effect, R uses the function `Vectorize`, and Python uses `map` or a list assembly.

4. Module StatsBase.jl overview

In the module `StatsBase.jl` [10] implement basic functions for working with statistical samples presented as one-dimensional arrays. Due to the ease of use of most functions, we will not dwell on examples, but will give only short description of the main functionality of this module.

— Vectors of the sample weight coefficients (weight vectors).
— Functions that calculate the mean (geometric, harmonic, power and weighted arithmetic mean).
— The simplest statistical functions.
  – Moments that take into account the vectors of weight coefficients: mathematical expectation, variance, standard deviation, skewness coefficient, kurtosis coefficient and central moments of arbitrary order.
  – Standardized score (Z-score).
  – Entropy calculations, such as standard, Rényi (generalized) entropy, crossentropy, Kullback-Leibler divergence distance.
  – Quantiles and mods.
— Robust statistics: truncation and winsorization of the sample.
— Comparing two samples, by calculating different discrete metrics.
— Calculation of scattering, covariance, and correlation matrices.
— Functions that calculate the frequency of occurrence of a particular value in the sample.
— Calculation of histograms.
— Autocorrelation and autocovariance.

Functions from the module `StatsBase.jl` is actively used in other modules, so it is included in the list of dependencies of most statistical libraries created for `Julia`.

5. Module Distributions.jl overview

5.1. Brief overview of the module

The module `Distributions.jl` [21] implements functions and methods related to probability distributions (mainly one-dimensional discrete and continuous, as well as a small number of multidimensional ones).

— Probability distribution Functions (CDF) and probability distribution density functions (PDF).
— Functions for calculating statistical characteristics of distributions (expectation, variance, moments, modes, quantiles, kurtosis, etc.).
— Characteristic functions of distributions and generating functions of moments.
— Methods for selecting distribution parameters based on statistical data (distribution fitting) by the maximum likelihood method (Maximum Likelihood) and the Sufficient Statistics method (Sufficient Statistics).

5.2. Module installation

In order to use the module `Distributions.from`, it must first be installed using the command `Pkg.add("Distribution.from")`. After that, it can be imported using the instructions `using` or `import`. We use the second method to avoid mixing the module namespaces `Distributions.jl` with the global scope.

```
import Distributions
const dist = Distributions
```

Now `dist` will serve as a short synonym for Distribution and all functions and variables defined in the module name area will be accessible via the period `operator.`.

5.3. Creating a probability distribution

Since Julia's custom data types are not inferior in performance to the built-in data types, in the module `Distributions.jl`, probability distributions are implemented as additional data types. For example, to set a normal distribution, you should call the constructor `Normal`, passing to it two parameters: μ and σ are the mathematical expectation and the standard deviation:

```
μ, σ = 0.0, 1.0 # location, scale
Normal = dist.Normal(μ, σ)
```

The variable `Normal` will now have the type `Distributions.Normal{Float64}`. If you call the constructor without arguments, then the standard normal distribution will be set, with $\mu = 0$ and $\sigma = 1$.

Similarly, other distributions can be set, for example, the Beta distribution:

```
# Beta distribution
α, β = 1.0, 1.0 # shape
Beta = dist.Beta(α, β)
```

Complete lists of discrete and continuous distributions are given in the tables 1 and 2. On the official documentation page [11] there is a description of almost all implemented distributions, from which you can find out what parameters and in what order you need to pass to the constructor and what default values are provided for these parameters. Basically, developers adhere to the established notation in the literature.

Table 1
List of one-dimensional discrete distributions implemented in the module `Distribution.jl`

№	Function	Description
1	`Bernoulli`	Bernoulli distribution
2	`BetaBinomial`	Beta-Binomial distribution
3	`Binomial`	Binomial distribution
4	`Categorical`	Categorical distribution
5	`DiscreteUniform`	Discrete Uniform distribution
6	`Geometric`	Geometric distribution
7	`Hypergeometric`	Hypergeometric distribution
8	`NegativeBinomial`	Negative Binomial distribution
9	`Poisson`	Poisson distribution
10	`PoissonBinomial`	Poisson-Binomial distribution
11	`Skellam`	Skellam distribution

Table 2
A complete list of distributions implemented in the `Distribution.jl` module and similar functions from the `scipy.stats` library from various modules of the `R` language

No	Distribution	Function name		
		`Distributions.jl`	`scipy.stats`	`R`
1	Arcsine	`Arcsine`	`arcsin`	`[distr]`
2	Beta	`Beta`	`beta`	`beta`
3	Beta Prime	`BetaPrime`	`betaprime`	`{VGAM}`
4	Biweight	`Biweight`	-	-
5	Cauchy	`Cauchy`	`cauchy`	`cauchy`
6	Chi	`Chi`	`chi`	`[Runuram]`
7	Chisq	`Chisq`	`chi2`	`chisq`
8	Cosine	`Cosine`	`cosin`	-
9	Erlang	`Erlang`	`erlang`	`gamma`

Table 2

A complete list of distributions implemented in the `Distribution.jl` module and similar functions from the `scipy.stats` library from various modules of the `R` language (continuation)

No	Distribution	Function name		
		`Distributions.jl`	`scipy.stats`	`R`
10	Epanechnikov	`Epanechnikov`	-	`[epandist]`
11	Exponential	`Exponential`	`expon`	`exp`
12	Fisher Distribution	`FDist`	`f`	`f`
13	Frechet	`Frechet`	`frechet_r`, `frechet_l`	`[VGAM]`
14	Gamma	`Gamma`	`gamma`	`gamma`
15	Generalized Extreme Value	`GeneralizedExtreme Value`	`genextreme`	`[spatial Extremes]`
16	Generalized Pareto	`GeneralizedPareto`	`genpareto`	`[evd]`
17	Gumbel	`Gumbel`	`gumbel_r`	`[VGAM]`
18	Inverse Gamma	`InverseGamma`	`invgamma`	`[extra Distr]`
19	Inverse Gaussian	`InverseGaussian`	`invgauss`	`[statmod]`
20	Kolmogorov	`Kolmogorov`	-	`[kolmim]`
21	K-S test	`KSDist`	`kswobign`	`[kolmin]`
22	K-S test one side	`KSOneSided`	`ksone`	`[kolmim]`
23	Laplace	`Laplace`	`laplace`	`[distr]`
24	Levy	`Levy`	`levy`	`[VGAM]`
25	Logistic	`Logistic`	`logistic`	`[VGAM]`
26	Log-Normal	`LogNormal`	`lognormal`	`lnorm`
27	Noncentral Beta	`NoncentralBeta`	-	`beta`
28	Noncentral Chisq	`NoncentralChisq`	`ncx2`	`chisq`
29	Noncentral Fisher	`NoncentralF`	`ncf`	`sadists`
30	Noncentral Student	`NoncentralT`	`nct`	`sadists`
31	Normal	`Normal`	`norm`	`norm`

Table 2
A complete list of distributions implemented in the `Distribution.jl` module and similar functions from the `scipy.stats` library from various modules of the `R` language (continuation)

No	**Distribution**	**Function name**		
		`Distributions.jl`	`scipy.stats`	`R`
32	NormalCanon	`NormalCanon`	-	`norm`
33	Normal Inverse Gaussian	`NormalInverseGaussian`	-	`[ghyp]`
34	Pareto	`Pareto`	`pareto`	`[VGAM]`
35	Rayleigh	`Rayleigh`	`rayleigh`	`[VGAN]`
36	Wigner semicircle distribution	`Semicircle`	`semicircular`	
37	SymTriangu-larDist	`SymTriangularDist`	-	`[triangle]`
38	Student	`TDist`	`t`	`t`
39	Triangu-larDist	`TriangularDist`	`triang`	`[triangle]`
40	Triweight	`Triweight`	-	-
41	Uniform	`Uniform`	`uniform`	`unif`
42	Von Mises	`VonMises`	`vonmises`	`[mov MF]`
43	Weibull	`Weibull`	`weibull_min`, `weibull_max`	`weibull`

5.4. Calculation distributions characteristics

A number of functions listed in the table 3 are intended to obtain characteristics of theoretical distributions. As an argument, they take a variable of type `Distribution` and, depending on the distribution, return the requested parameters. If this type of distributions does not have one or another parameter, an exception is thrown. For example, the `dof` function returns the number of degrees of freedom of the distribution, so that for a normal distribution it will end with an exception thrown, and for a Student distribution it will return the value of the requested parameter.

Another set of functions is used to calculate statistical characteristics of both theoretical distributions and empirical data. A complete list of these functions is given in the table 4. Some of them are defined in the scope of the Julia language base module (`Base`), as they overload standard functions such as `mean`, `median`, etc. More specific functions are defined in the scope of the module `Distributions`.

A number of functions can be used to calculate the statistical characteristics of an empirical sample (i.e., take an array of random numbers as the first argument). For such functions in the table 4 in the column `v` there is a mark `+`. Note that the function `quantile` supports an array as the first argument, while the function `cquantile` does not.

Table 3
List of functions for getting distribution parameters from the module `Distribution.jl`

Function	**Description**
`params(d)`	return description's parameters
`succprob(d)`	not implemented yet
`failprob(d)`	not implemented yet
`scale(d)`	return `scale` parameter (if not, throw error)
`location(d)`	return `location` parameter (if not, throw error)
`shape(d)`	return `shape` parameter (if not, throw error)
`rate(d)`	return `rate` parameter (if not, throw error)
`ncategories(d)`	return number of categories
`ntrials(d)`	get the number of trials
`dof(d)`	return degree of freedom

Consider an example. So the function `mean` can be used to find the mathematical expectation if one passes as an argument an object representing some distribution. Or to calculate the average, if one passes an array of numbers (representing a statistical sample).

```
mean(Normal) # return 0
mean([1, 2, 3, 4, 5]) # return 3
```

5.5. One dimensional distributions

In the Distributions package.jl implemented 11 discrete distributions (table 1) and 43 continuous one-dimensional distributions (see table 2). This table gives a summary of the functions from Distributions.jl and its equivalent functions from the library `scipy.stats` and language packages `R`. This will allow readers familiar with `R` or `scipy.stats` to orient themselves.

5.6. Implementations of functions for calculating statistical characteristics for different distributions

The table 5 shows the results of testing functions that calculate statistical characteristics for various distributions. In the table, the sign `+` means that the function is implemented for this distribution, and the sign `-` means that there is no implementation. Note that these are implementations specifically for theoretical distributions. It should also be borne in mind that the lack of implementation may mean that there is no exact analytical formula for this distribution.

Table 4
List of functions for calculating statistical characteristics of distributions from the module `Distribution.jl`

Function	**Description**	**V**
`maximum(d)`	maximum value	+
`mininum(d)`	minimum value	+
`mean(d)`	mathematical expectation	+
`var(d)`	variation	+
`std(d)`	standard deviation	+
`median(d)`	median	+
`dist.mode(d)`	mode	+
`dist.skewness(d)`	asymmetry coefficient	+
`dist.kurtosis(d)`	kurtosis coefficient	+
`dist.isplatykurtic(d)`	checks the kurtosis coefficient (> 0, < 0, $= 0$)	-
`dist.isleptokurtic(d)`		-
`dist.ismesokurtic(d)`		-
`dist.entropy(d)`	entropy	+
`dist.pdf(d, t)`	PDF	-
`dist.cdf(d, t)`	CDF	-
`dist.logpdf(d, t)`	ln from PDF	-
`dist.logcdf(d, t)`	ln from CDF	-
`dist.mgf(d, t)`	generating function of moments	-
`dist.cf(d, t)`	characteristic distribution function	-
`quantile(d, q)`	quantile q	+
`cquantile(d, q)`	complementary quantile $1 - q$	-
`invlogcdf(d, t)`	inverse function for logcdf	-
`invlogccdf(d, t)`	inverse function for logccdf	-

Table 5

Implementation of functions for calculating statistical characteristics of distributions from the module `Distribution.jl`.
(Notation: `+` — method is implemented, `-` — method is not implemented (MethodError), `@` — domain error, `o` — any other error)

`maximum`	`minimum`	`mean`	`var`	`std`	`median`	`mode`	`skewness`	`kurtosis`	`isplatykurtic`	`isleptokurtic`	`ismesokurtic`	`entropy`	`pdf`	`cdf`	`logpdf`	`logcdf`	`mgf`	`cf`	`quantile`	`cquantile`	`invlogcdf`	`invlogccdf`	Distribution
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-	+	+	+	+	Arcsine
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-	+	+	+	+	Beta
+	+	+	+	+	+	+	+	-	-	-	-	-	+	+	+	+	-	-	+	+	+	+	Beta Prime
+	+	+	+	+	+	+	+	+	+	+	+	-	+	+	+	+	+	+	+	+	+	+	Biweight
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	Cauchy
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-	+	+	+	+	Chi
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	Chisq
+	+	+	+	+	+	+	+	+	+	+	+	-	+	+	+	+	-	-	+	+	o	o	Cosine
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	Erlang
+	+	+	+	+	+	+	+	+	+	+	+	-	+	+	+	+	+	+	+	+	+	+	Epanechnikov
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	@	+	Exponential
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-	+	+	+	+	Fisher Distribution
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-	+	+	+	@	Frechet
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	Gamma
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-	+	+	+	@	Generalized Extreme Value

Table 5

Implementation of functions for calculating statistical characteristics of distributions from the module `Distribution.jl` (continuation)

`maximum`	`minimum`	`mean`	`var`	`std`	`median`	`mode`	`skewness`	`kurtosis`	`isplatykurtic`	`isleptokurtic`	`ismesokurtic`	`entropy`	`pdf`	`cdf`	`logpdf`	`logcdf`	`mgf`	`cf`	`quantile`	`cquantile`	`invlogcdf`	`invlogccdf`	Distribution
+	+	+	+	+	+	–	+	+	+	+	+	–	+	+	+	+	–	–	+	+	+	+	Generalized Pareto
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	–	–	+	+	@	@	Gumbel
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	@	+	+	+	+	+	Inverse Gamma
+	+	+	+	+	+	+	+	+	+	+	+	–	+	+	+	+	–	–	+	+	+	+	Inverse Gaussian
+	+	+	+	+	+	+	–	–	–	–	–	–	+	+	+	+	–	–	+	+	+	+	Kolmogorov
+	+	–	–	–	–	–	–	–	–	–	–	–	–	+	–	+	–	–	–	–	–	–	K-S
+	+	–	–	–	–	–	–	–	–	–	–	–	–	+	–	+	–	–	–	–	–	–	K-S one side
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	@	@	Laplace
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	@	Levy
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	@	@	Logistic
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	–	–	+	+	+	+	Log-Normal
+	+	–	–	–	+	–	–	–	–	–	–	–	+	+	+	+	–	–	+	+	+	+	Noncentral Beta
+	+	+	+	+	+	–	+	+	+	+	+	–	+	+	+	+	+	+	+	+	+	+	Noncentral Chisq
+	+	+	+	+	+	–	–	–	–	–	–	–	+	+	+	+	–	–	+	+	+	+	Noncentral Fisher
+	+	+	+	+	+	–	–	–	–	–	–	–	+	+	+	+	–	–	+	+	+	+	Noncentral Student

Table 5

Implementation of functions for calculating statistical characteristics of distributions from the module `Distribution.jl` (continuation)

`maximum`	`minimum`	`mean`	`var`	`std`	`median`	`mode`	`skewness`	`kurtosis`	`isplatykurtic`	`isleptokurtic`	`ismesokurtic`	`entropy`	`pdf`	`cdf`	`logpdf`	`logcdf`	`mgf`	`cf`	`quantile`	`cquantile`	`invlogcdf`	`invlogccdf`	Distribution
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	Normal
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	–	–	+	+	+	+	NormalCanon
+	+	+	+	+	–	–	+	+	+	+	+	–	+	–	+	–	–	–	–	–	–	–	Normal Inverse Gaussian
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	–	–	+	+	+	+	Pareto
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	–	–	+	+	@	@	Rayleigh
+	+	+	+	+	+	+	+	–	–	–	–	+	+	+	+	+	–	–	+	+	+	+	Wigner semicircle distribution
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	@	@	SymTriangularDist
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	–	+	+	+	+	+	Student
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	@	@	TriangularDist
+	+	+	+	+	+	+	+	+	+	+	+	–	+	+	+	+	+	+	+	+	+	+	Triweight
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	Uniform
+	+	+	+	+	+	+	–	–	–	–	–	+	+	+	+	+	–	+	–	–	–	–	Von Mises
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	–	–	+	+	@	+	Weibull

For those distributions for which at least one of the functions PDF and CDF is implemented, graphs are constructed (see, for example in figures 1 and 2).

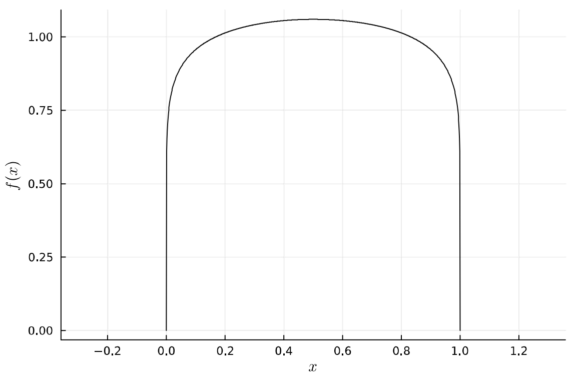

Figure 1. Example of Beta distribution PDF

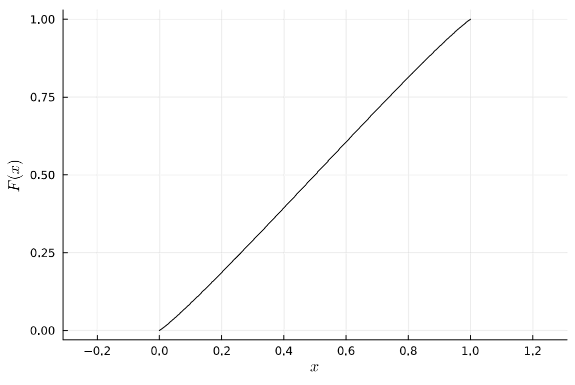

Figure 2. Example of Beta distribution CDF

5.7. The distribution parameters determination by the sample

To determine the parameters of the theoretical distribution over the sample, you can use the universal function `fit(d, x)`.

This function uses either the maximum likelihood method or the method of sufficient statistics in its work. The first method is implemented in the function `fit_me`, and the second one in the function `suffstats`. The developers recommend using the function `fit`, which chooses the optimal method itself. Consider an example of usage.

```
X = rand(Normal, 100)
# A normal distribution whose coefficients
# calculated based on a statistical sample
E_Normal = dist.fit(dist.Normal, X)
```

Selection of coefficients based on the sample is implemented only for a small number of distributions available in the module. Among which:

— Bernoulli distributions, discrete uniform, geometric, binomial, categorical and Poisson distributions;
— Beta, exponential, normal, gamma, Laplace, Pareto, uniform distributions.

As you can see, among the distributions there are no quite commonly used ones, such as the Weibull distribution or the lognormal distribution.

6. Conclusion

As a result of the review of the capabilities of the Julia language in the field of mathematical statistics, it can be concluded that in terms of the richness of functionality, it is still inferior to the specialized R language and the capabilities of Python libraries. However, it surpasses these languages in the speed factor, and the intensity of the Julia language development makes it possible to assume that the missing functionality will be implemented over time.

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] B. Lauwens and A. Downey, *Think Julia.* O'Reilly Media, Inc., 2019.

[3] T. Kwong, *Hands-on design patterns and best practices with Julia.* Packt Publishing, 2020.

[4] C. T. Kelley, *Solving nonlinear equations with iterative methods, Solvers and Examples in Julia.* SIAM, 2022.

[5] J. Bezanson, J. Chen, B. Chung, S. Karpinski, V. B. Shah, J. Vitek, and L. Zoubritzky, "Julia: dynamism and performance reconciled by design," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–23, Oct. 2018. DOI: `10.1145/3276490`.

[6] M. N. Gevorkyan, A. V. Korolkova, D. S. Kulyabov, and K. P. Lovetskiy, "Statistically significant comparative performance testing of Julia and Fortran languages in case of Runge–Kutta methods," in *Numerical methods and applications. NMA 2018*, ser. Lecture Notes in Computer Science, G. Nikolov, N. Kolkovska, and K. Georgiev, Eds., vol. 11189, Cham: Springer International Publishing, 2019, ch. 45, pp. 400–407. DOI: `10.1007/978-3-030-10692-8_45`.

[7] "JuliaStats, Statistics and machine learning made easy in julia." (2023), [Online]. Available: `https://juliastats.org/`.

[8] Y. Nazarathy and H. Klok, *Statistics with Julia, Fundamentals for Data Science, Machine Learning and Artificial Intelligence.* Springer International Publishing, 2021. DOI: `10.1007/978-3-030-70901-3`.

[9] "Julia forums." (2023), [Online]. Available: `https://discourse.julialang.org`.

[10] "StatsBase.jl." (2023), [Online]. Available: `https://github.com/JuliaStats/StatsBase.jl`.

[11] "Distributions.jl." (2023), [Online]. Available: `https://github.com/JuliaStats/Distributions.jl`.

[12] C. Führer, J. E. Solem, and O. Verdier, *Scientific computing with Python, High-performance scientific computing with NumPy, SciPy, and pandas*, 2nd. Packt Publishing Ltd., 2021.

[13] D. Toomey, *Learning Jupyter.* Packt Publishing Ltd., 2016.

[14] "Pluto.jl — interactive Julia programming environment." (2023), [Online]. Available: `https://plutojl.org/`.

[15] “Windows terminal, console and command-line repo.” (2023), [Online]. Available: `https://github.com/microsoft/terminal`.

[16] “Miniconda.” (2023), [Online]. Available: `https://docs.conda.io/en/latest/miniconda.html`.

[17] “Pluto.jl GitHub.” (2023), [Online]. Available: `https://github.com/fonsp/Pluto.jl`.

[18] “Julia GitHub.” (2023), [Online]. Available: `https://github.com/JuliaLang/julia`.

[19] A. V. Korolkova, M. N. Gevorkyan, and D. S. Kulyabov, “Implementation of hyperbolic complex numbers in Julia language,” vol. 30, no. 4, pp. 318–329, Dec. 2022. DOI: `10.22363/2658-4670-2022-30-4-318-329`.

[20] 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`.

[21] M. Besançon, T. Papamarkou, D. Anthoff, A. Arslan, S. Byrne, D. Lin, and J. Pearson, “Distributions.jl: Definition and modeling of probability distributions in the JuliaStats ecosystem,” *Journal of Statistical Software*, vol. 98, no. 16, pp. 1–30, 2021. DOI: `10.18637/jss.v098.i16`.

For citation:

M. N. Gevorkyan, A. V. Korolkova, D. S. Kulyabov, Julia language features for processing statistical data, Discrete and Continuous Models and Applied Computational Science 31 (1) (2023) 5–26. DOI: 10.22363/2658-4670-2023-31-1-5-26.

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. — 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: `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 537.8:512.723
DOI: 10.22363/2658-4670-2023-31-1-5-26
EDN: VNJCSU

Возможности языка Julia для обработки статистических данных

М. Н. Геворкян[1], **А. В. Королькова**[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, д. 6, Дубна, Московская область, 141980, Россия*

Аннотация. Язык программирования Julia является специализированным языком для научных вычислений. Язык сравнительно новый, поэтому большинство библиотек для него находится в активной стадии разработки. В статье авторы рассматривают возможности применения языка в области математической статистики. Особый акцент делается на технической составляющей, в частности подробно описывается процесс установки и настройки программного окружения. Так как пользователи языка Julia зачастую не являются профессиональными программистами, технические моменты в настройке программного окружения могут вызывать у них трудности, препятствующие быстрому освоению базовых возможностей языка. В статье описываются некоторые особенности Julia, которые отличают его от других популярных языков, используемых для научных вычислений. Также даётся обзор двух основных библиотек для математической статистики. Упор опять-таки делается на технической стороне, чтобы дать читателю представление об общих возможностях языка в области математической статистики.

Ключевые слова: язык программирования Julia, обработка статистических данных