Contents

1 Introduction

In the world of today's sciences, the use of computers has transformed the field and requirements for researchers tremendously. The ability to collect and process data at high rates using (semi-) automatic approaches pushes the classical research fields like physics, biology, medicine, engineering, and social sciences towards hybrid models and big data evaluation. A recent example is the detection of gravitational waves, where multiple software components have been used to process the data and help the researchers to classify the events [13] .

The requirements on software development used for research purposes have decreased below expert knowledge. Research software and coding are much more of a general tool, similar to text editors, for many scientific disciplines. As a result scientists from fields where programming and numerical computation play at most a minor role, e.g. psychology, sociology but also biology, start to develop code on their own. This is reflected in the rise of high-level languages like *Python*, *Ruby*, *R*, or *Lua* [22, 20, 17, 5] which are semantically easier to program as opposed to lower-level languages such as *C*, *C++*, *Fortran*, *Go*, and *Rust* [9, 19, 1, 16, 15]. The broader, more diverse user base of high level languages results in a more versatile ecosystem, giving rise to various packages and modules designed for specific tasks with a clean interface. Despise these advantages, the price to be paid is commonly linked to the performance in terms of execution time and memory consumption of the resulting code. A common pattern used is hence to write an application programming interface (API) in high-level languages and move the computationally expensive parts to a low-level implementation after finishing prototyping. This is referred to as the *two language problem*. This approach, while being general, has some major drawbacks related to datatype availability, effort of maintanance, and dependency management - to name just a few.

To resolve this - and other - problems of scientific computation the *Julia* programming language has been created [2]. It aims to provide performant code, leveraging the best features of prominent languages without complicated semantic structures, contradicting the common misconception that computational performance comes at the price of ease of use [3]. Even though being a relatively new language, initially released in 2012 for the broader public, *Julia* gained much momentum.

In the following, *Julia* will be introduced as a language for scientifc computation. Section 2 starts by introducing how to define and analyse functions before introducing the concept of multiple dispatch to the reader. Afterward, section 3 explains benchmarking, optimizing code, and just-in-time compilation. Section 4 a simple greedy algorithm for sparse regression is implemented, showcasing the ease of use to transpile mathematical notation to fast code. Section 5 highlights some of the most useful and prominent packages of the language. A conclusion is drawn in section 6.

2 Basics and Multiple Dispatch

There exist different ways to define functions in Julia. Similar to e.g. Python or Matlab, a simple Gaussian radial basis function $f : \mathbb{R} \times \mathbb{R} \mapsto \mathbb{R}$ can be implemented as follows

```
# Define a function
f(x,y) = exp(-(x-y)^2)

# Evaluate the function
f(0.2, 1) # Returns 0.5272924240430484

# Evaluate with a Float32
f(0.2f0, 1) #Returns 0.5272924f0
```

As we can see, we defined no input types for the function and called it with a Float64 and Int64 as arguments, resulting in a Float64 as the best intersection of both number types. Changing the input type of the first argument to Float32 returns Float32, again using the intersection of the input data types. To see how the evaluation works, we will use

```
@code_typed f(0.2, 1)
```

Which will evaluate the function and returns a lowered and type-inferred abstract syntax tree for the method

```
CodeInfo(
    1 ─ %1 = Base.sitofp(Float64, y)::Float64
    │   %2 = Base.sub_float(x, %1)::Float64
    │   %3 = Base.mul_float(%2, %2)::Float64
    │   %4 = Base.neg_float(%3)::Float64
    │   %5 = invoke Main.exp(%4::Float64)::Float64
    └──      return %5
    ) => Float64
```

The above output contains two important informations: the intermediate representation (IR), indicated by the lines with a leading %, passed on to the compiler and the type information propagated through the function. It first the value of y is converted into a Float64 (%1), the difference $x - y$ is computed (%2), multiplied with itsself (%3), negated (%4) and mapped via the exponential function (%5). Entering different arguments also works out of the box. Using a pair of ComplexFloat64 and Float32 returns a complex-valued result, as can be expected. Different representations as the one shown above can be accessed via @code_lowered, @code_typed, @code_lowered and @code_native. These return the IR, IR with type information, the code for the compiler and the machine code respectively.

```
f(im*0.2, 1f0) # 0.3526677021528701 + 0.1491055129382033im
```

The reason why this approach works is called multiple dispatch (MD) and a core design element of *Julia*. Most prominent languages operate on obect-oriented program-

	FP	OOP	MD
Dispatch arguments	$\emptyset$	$\{x_1\}$	$\{x_1, x_2, \dots, x_N\}$
Expressive Order	$\mathcal{O}(1)$	$\mathcal{O}(\lvert x_1 \rvert)$	$\mathcal{O}\left(\prod_{i=1}^{N} \lvert x_i \rvert\right)$
Expressive Power	const.	linear	exponential

Table 1: Expressiveness of different programming paradigms

ming (OOP) or use functional programming[1] (FP). In FP, each function is defined uniquely by its name and stored in the global namespace. OOP stores methods related to objects in the namespace of the corresponding object definition, allowing for more expressiveness and clarity in defining the functions.

Following [8], the expressiveness of a language can be defined as the number of different methods with the same name but different purposes depending on their arguments. Consider the exponential function $exp : \mathbb{X} \mapsto \mathbb{X},\ \mathbb{X} \in \{\mathbb{R}, \mathbb{C}, \dots\}$. FP would define several methods by name for different arguments, e.g. $exp_{\mathbb{R}}$, $exp_{\mathbb{C}}$. In OOP, the object itself is the dispatch argument, so a pseudo-code notation would result in $\mathbb{R}.exp$, $\mathbb{C}.exp$. In MD, all input arguments are checked, hence *exp* can be used and extended for all types it holds a definition for. These insights are summarized in Table 1, showing the proportional growth of expressiveness in relation to their dispatch arguments.

If a general function, e.g. +, is called in *Julia*, the compiler automatically uses the right function definition based on the arguments. This can be confirmed with

```
methods(+, (Number,Number))
# 40 methods for generic function "+":
[1] +(x::T, y::T) where T<:Union{Int128, ..., UInt8} in Base at int.jl:87
[2] +(c::Union{UInt16, UInt32, UInt64, UInt8}, x::BigInt) in Base.GMP at gmp.jl:528
[3] +(c::Union{Int16, Int32, Int64, Int8}, x::BigInt) in Base.GMP at gmp.jl:534
[4] +(c::Union{UInt16, UInt32, UInt64, UInt8}, x::BigFloat) in Base.MPFR at mpfr.jl:376
[5] +(c::Union{Int16, Int32, Int64, Int8}, x::BigFloat) in Base.MPFR at mpfr.jl:384
[6] +(c::Union{Float16, Float32, Float64}, x::BigFloat) in Base.MPFR at mpfr.jl:392
...
```

which returns all methods defined in the current workspace for the + operator using two Number types as inputs. Number represents an abstract type spanning various number types, e.g. Float, Int, UInt, Complex. Evaluating the following code block

```
# Infer which function is called
@which exp(0.2f0)
@which exp(im*0.2f0)
```

returns the corresponding method used for computation and where it is defined.

```
exp(x::T) where T<:Float32 in Base.Math at special/exp.jl:229
exp(z::Complex) in Base at complex.jl:638
```

A key difference to OOP based programming is that no method definition has

[1]FP refers to purely functional programming in the scope of this work. In general modern languages using FP as a paradigm, like *Closure*, tend to use MD.

to explicitly know of the other, as long as it is unique in its dispatch arguments[2]. This allows developers to easily extend existing packages and functionalities, allowing excessive reuse of the codebase[3]. As a small example consider defining a new Number type and its corresponding exp:

```
using Base: Number
struct MyNumber <: Number end

# Dispatch on the definition
Base.exp(t::MyNumber) = println("Foo")

# Execute
a = MyNumber()
exp(a) # Returns "Foo"
```

To make the function *f* useable for vector operations, multiple dispatch can be leveraged as well:

```
# Import Linear Algebra
using LinearAlgebra

# Add a function dispatch on two vectors
f(x::AbstractVector, y::AbstractVector) = exp.(-(x-y).^2)

# Define two random vectors
x = randn(10); y = randn(10);

f(x,y) # Returns a vector
```

Since both exp and ^ are not defined for vectors, we use broadcasting to call the function on each element of the data structure individually. This is accomplished via the . operator. Checking the available definitions of the function via

```
methods(f) # Show the available methods corresponding to f
```

we can see that two definitions are present

```
# 2 methods for generic function "f":
[1] f(x::AbstractVector{T} where T, y::AbstractVector{T} where T) in Main at ./HelloWorld.jl:32
[2] f(x, y) in Main at ./HelloWorld.jl:3
```

[1] is the definition we just added and [2] our original implementation from the beginning of this section, extending the function to be reused and avoiding different naming schemes.

[2]An unwanted dispatch on an already defined method can result unstable behaviour and called *type-piracy*

[3]A good example is given by `https://github.com/JuliaPlots/RecipesBase.jl`, which is used to define individual recipes used for plotting without explicitly depending on *Plots.jl*

3 Just-In-Time Compilation and Optimizing Code

A common complain of users new to *Julia* is that is does not hold up to its promise of speed, often feels even slower than e.g. using *Python* for the same task. This chapter gives a brief explanation on why this is the case, how performance should and can be measured and how to optimize code for performance. Consider the following code

```
using LinearAlgebra

g(x, y) = x + sin(x)*y^2

function g(x::Vector, y::Vector)
    z = similar(x)
    for i in 1:length(x)
        z[i] = g(x[i], y[i])
    end
    return z
end

x = randn(10); y = randn(10);

@time g(x,y)
```

Here the @time macro[4] returns basic performance metrics about the function call like overall execution time, memory allocations, and compilation time.

```
    0.016432 seconds (6.08 k allocations: 342.705 KiB, 99.78% compilation time)
```

Another execution of the same command holds a different result.

```
    0.000008 seconds (1 allocation: 160 bytes)
```

The reason is *Julia*'s underlying just-in-time (JIT) compilation. On the first call of the method g, the code is lowered to machine instructions and stored in memory for the specific types of arguments. This means that the first call of a method has an additional overhead due to the effort of the LLVM [12] compiler[5]. The second call of a method leverages the already compiled function and can be executed and benchmarked faster. A slight change of arguments shows that this process is indeed type-specific.

```
# Use another input vector of type Float32
z = randn(Float32, 10);

@time g(z,y)
```

Again, the compilation time is present and takes up most of the execution time

```
    0.100689 seconds (269.78 k allocations: 15.753 MiB, 10.07% gc time, 99.92% compilation time)
```

[4]A macro transforms the abstract syntax tree of the program. Interested readers might also refer to [11].

[5]Which is also the reason for the infamous time to first plot benchmark in *Julia*

JIT compilation also exists for different languages, such as *Python*, *lua* or *Matlab*[14], as an optional extension. However, most of the performance comes from compiler optimization where *Julia* exceeds performance-wise due to JIT being the standard. A slightly outdated benchmark comparing different algorithms and languages can be found on the micro-benchmark page.

Julia is storing array-like data in a column major memory layout, which highly impacts the speed. The execution of the following code shows that a column major iteration can result in a high speedup, using the same array.

```julia
## Memory Layout
using BenchmarkTools

function sum_by_row(x::AbstractMatrix)
    res = zero(eltype(x))
    for i in 1:size(x, 1)
        res += sum(x[i,:])
    end
    return res
end

function sum_by_col(x::AbstractMatrix)
    res = zero(eltype(x))
    for i in 1:size(x, 2)
        res += sum(x[:,i])
    end
    return res
end

x = randn(Float32, 10_000, 10_000);

@btime sum_by_row($x) # 713.33 ms (20000 allocations: 382.23 MiB)
@btime sum_by_col($x) # 99.842 ms (20000 allocations: 382.23 MiB)
```

Instead of using @time the performance metrics are derived via @btime [6], which evaluates the method over multiple trials, limited by the memory and time consumption of the function.

To investiage the use of parallel processing, consider the example given in [18, p. 178 ff.]

```julia
function sum_vectors!(x, y, z)
    n = length(x)
    for i in 1:n
        x[i] = y[i] + z[i]
    end
end

function sum_vectors_simd!(x, y, z)
    n = length(x)
    @inbounds @simd for i in 1:n
        x[i] = y[i] + z[i]
    end
end

a = zeros(Float32, 100_000);
b = randn(Float32, 100_000);
```

[6]BenchmarkTools.jl

```
c = randn(Float32, 100_000);

@btime sum_vectors!($a,$b,$c) # 79.932 μs (0 allocations: 0 bytes)
@btime sum_vectors_simd!($a,$b,$c) # 18.960 μs (0 allocations: 0 bytes)
```

Both methods defined above are mutating, indicated by the exclamation mark at the end of the function. They modify the values of their first argument, the array x, overwriting it inplace with no further allocations. We see that the use of the second function, which leverages single-instruction-multiple-data (SIMD) - via @simd - and disableing the checking of bounds- via @inbounds - holds a relative speedup of 4.18 in comparision to the first implementation. A similar syntax enables multithreading

```
using Base.Threads

function sum_vectors_threading!(x, y, z)
    n = length(x)
    @inbounds Threads.@threads for i in 1:n
        x[i] = y[i] + z[i]
    end
end

@btime sum_vectors_threading!($a,$b,$c) # 132.019 μs (6 allocations: 560 bytes)
```

Which shows the ease of use, but the communication time dominates the execution time and no speedup is achieved. Likwise, computations can be moved from the CPU to the GPU using different pacakges [18, p.204 ff.].

4 Orthogonal Matching Pursuit

In the following section, preconditioned generalized orthogonal matching pursuit (GOMP) [21] will be implemented. Matching pursuit algorithms are a class of greedy algorithms designed to solve the sparse signal recovery problem

$$\begin{aligned} \min_x \quad & \|x\|_0 \\ \text{s.t.} \quad & y = \Psi x \end{aligned} \tag{1}$$

where $x \in \mathbb{R}^n$ represents an unknown, K sparse basis for a signal $y \in \mathbb{R}^m$ in $\Psi \in \mathbb{R}^{m \times n}$, also called the sampling matrix. This is done by iteratively reducing the estimation error of the recovered signal by selecting S most similar, new components from the sampling matrix.

The main algorithm can be described as follows

```
Result: $x$
Data: $y$, $\Psi$, $K$, $S = 1$
Projection onto span;
$P = \Psi^T \left(\Psi\Psi^T\right)$; $\tilde{y} = Py$; $\tilde{\Psi} = P\Psi$;
Initialize residual and support;
$r = y$; $\Lambda = \emptyset$;
while Not converged do
    $\Omega = \delta_S\left(\left|\tilde{\Psi}^T r\right|\right)$;
    $\Lambda = \Lambda \cup \Omega$;
    $x = \min \left\|\tilde{y} - \tilde{\Psi}u\right\|_2 \quad , \quad supp(u) = \Lambda$;
    $r = \tilde{y} - \tilde{\Psi}x$;
end
;
```

Here $\delta_S(x) : \mathbb{R}^n \mapsto \{0,1\}^n$ denotes (with abuse of notation) the mapping of the S largest elements onto a corresponding indicator vector. A similar notation to the pseudocode above can be achieved in *Julia*. For brevity, only necessary elements of the source code will be shown, which can be found in detail in REF. The function is defined as follows

```julia
function gomp(y0::AbstractVector, Ψ::AbstractMatrix,
    K::Int = 2, S::Int = 1;
    max_iter::Int = 100, ϵ::Real = eps())
```

First, we will focus on the preconditioning via the matrix $P \in \mathbb{R}^{m \times m}$ mapping onto the column span of Ψ

```julia
    # Preconditioning
    P = ψ'pinv(ψ*ψ')
    # New matrix
    ψ = P*ψ
    y = P*y0
```

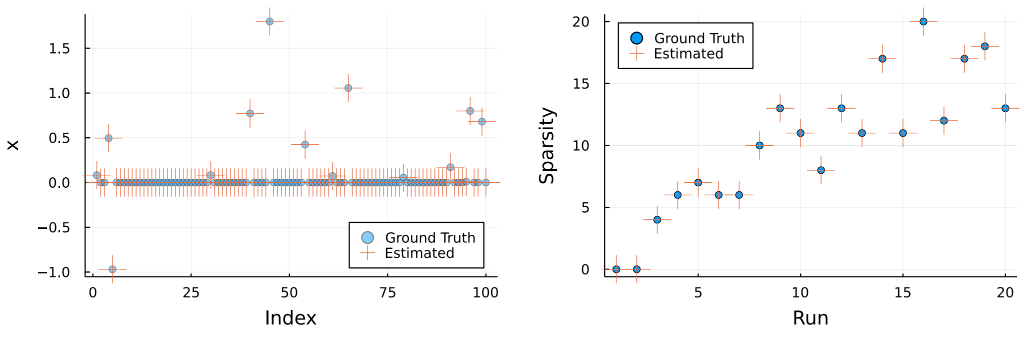

Figure 1: Performance of GOMP for a single sparse 100 dimensional vector (left) and with different sparsities (right). The signal y is of dimension 300 and has been generated by a random dense sampling matrix. GOMP has been used with $K = 50, S = 1$ and run for at most 100 iterations with a absolute tolerance $\epsilon = 0.1$.

Next, the iterative computation of the support, corresponding coefficients and residuals is performed

```
for i in 1:max_iter
    # Compute the similarities via magnitude
    amps .= ψ'r
    # Get the largest entry
    idx = sortperm(abs.(amps), rev = true)[1:S]
    # Update the support
    Λ[idx] .= true
    # Update the coefficients
    u[Λ] .= (y' / ψ')[1, Λ]
    # Update r
    r .= y - ψ*u
    # Convergence
    (norm(r,2) < ϵ || sum(Λ) >= K) && break
end
```

As can be seen from above, the implementation is quite similar to its mathematical counterpart, enhancing the interpretability and readability. The results of the algorithm are shown in Fig. 1, where a 100 dimensional sparse vector x has been recovered.

5 Ecosystem And Package Development

As mentioned in section 1 most high-level languages have a vast ecosystem of available packages, spanning various fields of application and a diverse userbase. Despite it being a relatively new language, *Julia* has 6068 officially registered open-source packages[7][8].

Due to the focus on scientific computation, we will start looking at packages related to linear algebra. LinearAlgebra is already included with the basic *Julia* installation, already providing BLAS and LAPACK access. MKL.jl provides an alternative backend for Intels MLK. Arpack.jl gives access to the arpack-ng library written in *Fortran*. IterativeSolvers.jl provides common algorithms for solving linear systems like MINRES, GMRES or Conjugate Gradients. GenericSVD.jl extends the singular value decomposition to special number types like BigFloats.

In the field of optimization, different packages like Optim.jl, Nonconvex.jl, or NLOpt.jl provide access to a variety of algorithms capable of optimizing multivariate functions written in or interfaced by *Julia*. For gradient and hessian-based algorithms, various automatic differentiation backends like ForwardDiff.jl, ReverseDiff.jl or Zygote.jl can be used in the absence of analytical expressions. A more structured approach is provided by the JuMP.jl organization, offering an domain-specific language (DSL) for rigorous modeling of optimization problems.

Machine learning is prominently represented by Flux.jl for (deep) neural networks. Next to SciKitLearn.jl similar libraries like MLJ.jl, Clustering.jl, and FastAI.jl offer common machine learning models.

One of the most influential domains in *Julia* is the numerical simulation domain, mainly the Scientific Machine Learning organization with 76[8] packages registered on JuliaHub. It provides tooling for simulation via DifferentialEquations.jl, neural ordinary differential equations, physics informed neural networks, and data-driven estimation. ModelingToolkit builds upon *Julia*'s symbolic computation libraries and offers acausal component-based modeling, automatic code optimization and provides a backend for many DSL like Catalyst.jl for chemical reactions.

While interoperability to *Fortran* and *C* is included in *Julia*, various packages like Cxx.jl, RCall.jl, PyCall, and CxxWrap.jl exist to wrap external code easily efficiently reducing the overhead of glue-code.
BinaryBuilder.jl allows the generation of precompiled binaries as dependencies via virtual machines automatically.

[7]JuliaHub
[8]July 29th, 2021

6 Conclusion

Julia set out to offer high-performance computation while providing clean and easily readable syntax, tackling the *two language problem* common in today's software. As this report shows, most of these promises have been fulfilled. Focusing on essential features for numerical linear algebra in the core installation, it readily provides necessary tools to conduct state-of-the-art research. MD allows for effective reuse and extension of methods and types, leading to a growing amount of packages that form a densely connected ecosystem with specialized scopes but familiar notation. Many of the algorithms provided reach high performance or are easily boosted via accessible code or type transformations via macros. While being fast and offering a versatile collection of algorithms on its own, *Julia* offers wrapper packages from and to several languages. This enables easy reuse of other sources and generation of APIs. While it can be used as a general programming language, most of the recently published research is still focused on classical fields like numerics, simulation, and control or physics, see e.g. [6, 4, 7].

An important, missing feature is the lack of support to generate independent executables and no hard real-time guarantees[9]. Exceptions, like [10], exist but are rare. In addition, *Julia* is still early in its maturation cycle, which still lowers its acceptance rate and userbase. *Python* became prominent around 2010, nearly 20 years after its initial release. This might increase the difficulties of many new users who have been trained to code in an OOP paradigm, which can make moving to *Julia* challenging.

The used code is available at GitHub in addition to the presentation.

[9]To the best knowledge of the author.

References

[1] John W Backus and William P Heising. “Fortran.” In: *IEEE Transactions on Electronic Computers* 4 (1964), pp. 382–385.

[2] Jeff Bezanson et al. “Julia: A Fresh Approach to Numerical Computing.” In: (July 2015). arXiv: `1411.1607 [cs]`.

[3] Alan Edelman. *A programming language to heal the planet together: Julia*. Youtube. 2019. URL: `https://youtu.be/qGW0GT1rCvs`.

[4] Marcelo Forets, Daniel Freire, and Christian Schilling. “Efficient reachability analysis of parametric linear hybrid systems with time-triggered transitions.” In: *2020 18th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. Dec. 2020, pp. 1–6. DOI: `10.1109/MEMOCODE51338.2020.9314994`.

[5] Roberto Ierusalimschy. *Programming in lua*. Roberto Ierusalimschy, 2006.

[6] Elias Jarlebring. “Broyden’s Method for Nonlinear Eigenproblems.” In: *SIAM Journal on Scientific Computing* 41.2 (2019), A989–A1012. DOI: `10.1137/18M1173150`. eprint: `https://doi.org/10.1137/18M1173150`. URL: `https://doi.org/10.1137/18M1173150`.

[7] Marek Kaluba, Dawid Kielak, and Piotr W. Nowak. “On property (T) for Aut(Fn) and SLn().” In: *Annals of Mathematics* 193.2 (2021), pp. 539–562. ISSN: 0003-486X. DOI: `10.4007/annals.2021.193.2.3`. URL: `https://www.jstor.org/stable/10.4007/annals.2021.193.2.3` (visited on 07/29/2021).

[8] Stefan Karpinski. *The Unreasonable Effectiveness of Multiple Dispatch*. Youtube. 2019. URL: `https://www.youtube.com/watch?v=kc9HwsxE1OY`.

[9] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. 2006.

[10] Twan Koolen and Robin Deits. “Julia for robotics: simulation and real-time control in a high-level programming language.” In: *2019 International Conference on Robotics and Automation (ICRA)*. ISSN: 2577-087X. May 2019, pp. 604–611. DOI: `10.1109/ICRA.2019.8793875`.

[11] Tom Kwong. *Hands-on design patterns and best practices with Julia : proven solutions to common problems in software design for Julia 1. x*. Birmingham, UK: Packt Publishing, 2020. ISBN: 183864881X.

[12] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis amp; Transformation.” In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO ’04. Palo Alto, California: IEEE Computer Society, 2004, p. 75. ISBN: 0769521029.

[13] Duncan M. Macleod et al. “GWpy: A Python package for gravitational-wave astrophysics.” en. In: *SoftwareX* 13 (Jan. 2021), p. 100657. ISSN: 2352-7110. DOI: `10.1016/j.softx.2021.100657`. URL: `https://www.sciencedirect.com/science/article/pii/S2352711021000029` (visited on 07/26/2021).

[14] MATLAB. *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[15] Nicholas D Matsakis and Felix S Klock II. "The rust language." In: *ACM SIGAda Ada Letters*. Vol. 34. 3. ACM. 2014, pp. 103–104.

[16] Jeff Meyerson. "The go programming language." In: *IEEE software* 31.5 (2014), pp. 104–104.

[17] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2016. URL: https://www.R-project.org/.

[18] Avik Sengupta. *Julia high performance : optimizations, distributed computing, multithreading, and GPU programming with Julia 1.0 and beyond*. Birmingham: Packt Publishing Ltd, 2019. ISBN: 178829811X.

[19] Bjarne Stroustrup. *The C++ programming language*. Upper Saddle River, NJ: Addison-Wesley, 2013. ISBN: 0321958322.

[20] David Thomas, Andrew Hunt, Chad Fowler, et al. *Programming Ruby: the pragmatic programmers' guide*. Raleigh, NC: Pragmatic Bookshelf, 2005.

[21] Zhishen Tong et al. "Preconditioned generalized orthogonal matching pursuit." In: *EURASIP Journal on Advances in Signal Processing* 2020.1 (May 2020), p. 21. ISSN: 1687-6180. DOI: 10.1186/s13634-020-00680-9. (Visited on 07/26/2021).

[22] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.