Scientific computing using the Julia programming language

Albert Piwonski[1], Damian Belz[1], Rodrigo Rezende[1], Stephanie Tchoumi[1]
[1] Project group at Technische Universität Berlin chair "Theoretische Elektrotechnik",
supervised by Mirsad Hadžiefendić and Marcus Christian Lehmann

A finite element method has been implemented using the Julia programming language (Julia PL). We restrict our investigations to the concepts and features of the Julia PL which are useful in computational electromagnetics in general. These are in particular Julia's type system, multiple dispatch and ability for parallel computing.

I. Introduction

Julia is a young programming language that has been designed especially for high performance numerical computing (cf. [1]). We present one model problem for using the Julia PL in the context of computational electromagnetics. More precisely, we implemented a finite element method for an electrostatic boundary value problem (BVP). Throughout we are rather focussing on the concepts and features of the programming language, than solving more complicated BVP's or choosing Ansatz functions of order > 1. We pay particular attention to Julia's type system, multiple dispatch and ability to perform parallel computing.

We organize the rest of the paper as follows: As a model problem, the BVP is formulated in the terminology of functional analysis in section II. Key steps of our finite element method implementation are sketched in section III. In section IV, we enlighten one main contribution of our paper: We argue which specific features of the Julia PL have been useful for a type stable and performant code. Therein is a brief study shown which compares the time difference between both serial and parallel implementations and the computational cost of them. A convergence study is investigated in section V. Our results are summarized in section VI, where we further discuss which future investigations could be worthwhile. [AP]

II. Model problem

Within the electrostatics regime, the electric field is curl-free, i.e. it can be expressed by a gradient of a scalar function ϕ, which is called electrostatic potential. If we assume that the permittivity behaves linear, isotropic and homogeneous – with respect to time and space – and the domain is free of charge one can derive Laplace's equation. We further assume a longitudinally symmetric problem, such that the computational domain $\overline{\Omega}$ is two dimensional, i.e. $\overline{\Omega} \subset \mathbb{R}^2$. Introducing Dirichlet conditions, which hold on the boundary $\partial\Omega := \Gamma_{D1} \cup \Gamma_{D2}$, leads to the elliptic boundary value problem $(\mathcal{B})$ whose domains are visualized in figure 1:

$$(\mathcal{B}) \begin{cases} \Delta\,\phi = 0 & \text{in } \Omega, \\ \phi = \phi_- & \text{on } \Gamma_{D1}, \\ \phi = \phi_+ & \text{on } \Gamma_{D2}. \end{cases}$$

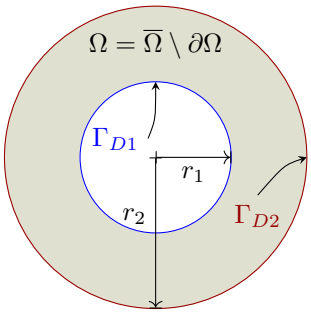

Fig. 1: Considered values for BVP $(\mathcal{B})$: $r_1 = 1\,\mathrm{m}$, $r_2 = 2\,\mathrm{m}$, $\phi_- = 0\,\mathrm{V}$, $\phi_+ = 1\,\mathrm{V}$.

It has to be mentioned that the function space of ϕ is the Sobolov space of square-integrable functions with weak derivatives of first order, i.e.:

$$\phi \in \mathrm{H}^1(\Omega) := \{u \in \mathrm{L}_2(\Omega) \,|\, \mathbf{grad}(u) \in \mathrm{L}_2(\Omega)^3\}, \text{ with} \quad (1)$$

$$\mathrm{L}_2(\Omega) := \{u \,|\, \int_\Omega u^2 \,\mathrm{d}\omega < \infty\}. \quad (2)$$

This information is necessary, because the finite element method in section III aims to approximate $\mathrm{H}^1(\Omega)$ with a finite dimensional subspace $\mathrm{S}_k \subset \mathrm{H}^1(\Omega)$. The BVP $(\mathcal{B})$ models an ideal cylinder capacitor, for which the analytic solution ϕ is known. This is beneficial for calculating errors considered in the convergence study in section V. [AP]

III. Implementation – Finite element method

This section describes our finite element method implementation for an approximation of the solution ϕ of BVP $(\mathcal{B})$.

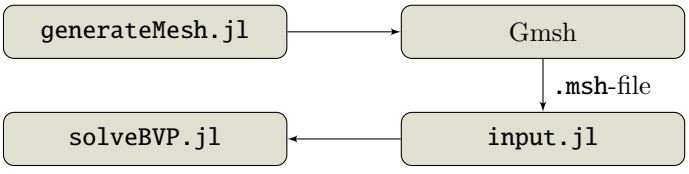

Fig. 2: Flow chart: Finite element method implementation. The source code is available at GitLab [7].

Figure 2 summarizes the main components of the implementation, which we briefly want to explain. The calculation of the stiffness matrix within `solveBVP.jl` is inspired by [4].

A. Mesh generation and choice of Ansatz functions

The Julia function `generateMesh.jl` uses the Gmsh API [3] for a generation of a partition of the computational domain $\overline{\Omega}$. First Γ_{D1} and Γ_{D2} have been defined by using the API commands `addPoint`, `addCircleArc` and `addCurveLoop`. With regard to the later incorporating of the DIRICHLET conditions, it is beneficial to define Γ_{D1} and Γ_{D2} as a physical group respectively, s.t. vertices with membership to $\partial\Omega$ can be extracted automatically. By usage of `addPlaneSurface` the inner computational domain, i.e. Ω, is defined. Calling `addPoint` with an additional parameter for the characteristic length γ leads to different resolved meshes, which are considered in chapters IV & V.

We restrict ourselves to the use of linear LAGRANGE elements, i.e. linear Ansatz functions on a triangles. These Ansatz functions are uniquely defined by the coordinates of the vertices of a triangle. Running `setOrder(1)` sets the degree of freedom (DoF) per element to $N = 3$. Executing `generate(2)` leads to a conformal DELAUNY triangulation $\mathcal{T}$ of $\overline{\Omega}$ (see figure 3). [AP]

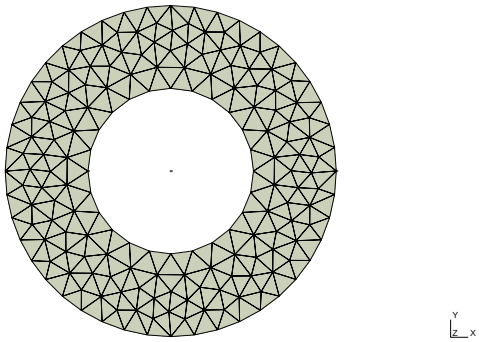

Fig. 3: Partition of the computational domain $\overline{\Omega}$: Function call of `generateMesh.jl` with $\gamma = 0.3\,\mathrm{m}$ leads to a conformal triangulation $\mathcal{T}$ with 385 elements.

B. System assembly

A discretised weak formulation of the boundary value problem ($\mathcal{B}$) can be described by the linear system of equations

$$A_{jk}\phi_k = b_j; \quad j,k = 1,...,N \quad \text{with} \tag{3}$$

$$A_{jk} = \sum_{T\in\mathcal{T}} \int_T \nabla\eta_i \cdot \nabla\eta_k \,\mathrm{d}x \quad \text{and} \tag{4}$$

$$b_j = -\sum_{k=1}^{N} \phi_k \sum_{T\in\mathcal{T}} \int_T \nabla\eta_i \cdot \nabla\eta_k \,\mathrm{d}x = -\sum_{k=1}^{N} \phi|_{\partial\Omega} A_{jk} \tag{5}$$

where $T \in \mathcal{T}$ is each a closed triangle and η_j are the linear Ansatz functions on triangles defined by $\eta_j(x_k, y_k) = \delta_{jk}$ as mentioned in III-A and [4]. For each element T, the entries of a local coefficient matrix M_{jk} are uniquely defined by the coordinates of the vertices of T as

$$M_{jk} = \int_T \nabla\eta_j (\nabla\eta_k)^{\mathrm{T}} \,\mathrm{d}x \tag{6}$$
$$= \frac{|T|}{(2|T|)^2} (y_{j+1} - y_{j+2}, x_{j+2} - x_{j+2}) \begin{pmatrix} y_{k+1} - y_{j+2} \\ x_{k+2} - x_{k+1} \end{pmatrix}$$

where $|T|$ is the area of the triangle. With $N = 3$ for T the 3×3 matrix M is symmetric and positive definite. The assembly of the global coefficient matrix A for a connected mesh as shown in figure 3 can be implemented in Julia with a loop over all elements in the generated mesh (for all $T \in \mathcal{T}$) with

```
for i in 1:size(msh.elements,1)
    A[msh.elements[i],msh.elements[i]]+=...
    ...stima(msh,msh.elements[i])
end.
```

The struct `msh::mesh` contains the reference to all coordinates of the corresponding triangulation $\mathcal{T}$ and all `msh.elements::Array{elements}`, which are implemented as tuples of indices.
`stima(msh,msh.elements[i])::Array{Float64,2}` represents the local coefficient matrix M and is calculated as shown in eqn. (6) for each given Triangle. The resulting A is sparse and symmetric and, as long as every discrete point is a vertex of at least one element, positive definite.

C. Boundary conditions

With $\Delta\phi = 0$ in Ω, the right hand side b depends only on the value of ϕ_- and ϕ_+ at the DIRICHLET boundaries. The known values for ϕ_k at the boundary vertices are assigned to the solution vector `u[k]::Float64` with the function `u_d1` and `u_d2` respectively. With the expression (5) a possible implementation in Julia is

```
u[inner_bound]=u_d2(inner_bound)
u[outer_bound]=u_d1(outer_bound)
b=b-A*u
```

where the API function `getNodesforPhysicalGroup` is previously used to extract the sets of vertices `inner_bound` and `outer_bound` as mentioned in III-A.

D. Solution of the linear system

In Julia, the linear solution for all free vertices can be performed with the backslash operator

```
u[freeN]=A[freeN,freeN]\b[freeN]
```

where the set of indices `freeN` is the relative complement of all vertex indices in the computational domain $\overline{\Omega}$ and the indices referencing to the vertices where ϕ_k is already defined by DIRICHLET conditions. [DB]

IV. IMPLEMENTATION – JULIA FEATURES

This section describes which specific Julia PL concepts and features have been useful within the implementation. We pay in particular attention to Julia's type system, multiple dispatch (IV-A) and ability to perform parallel computing (IV-B).

A. Typing in the Julia PL

In programming languages, a type system is a set of rules that assigns a property called type to the various constructs of a computer program, such as variables, expressions, functions or modules [9]. Julia's type system is dynamic, which means that the type of a value is unknown during compile-time and set during run-time, but gains some of the advantages of static type systems by making it possible to indicate that certain values are of specific types [10]. Declaring types within the code is done with the `::` Operator and:

- increases performance, because there is no need of a type inferencing process during run-time of the program, i.e. run-time is shifted to compile-time.
- can help to confirm that the program works as intended, by shifting run-time errors to compile-time errors.
- allows to use Julia's multiple-dispatch mechanism, i.e. adding multiple methods, which differ in the types of arguments, to the same function.

Compared to a statically typed language, where a type needs to be assigned to every expression, the Julia PL approach allows a more flexible workflow with the possibility to type code where it's needed.

1) Use-cases for type declaration

We briefly want to explain at which parts within the implementation we aimed static typing and where we deemed it appropriate to declare and assign user-defined types to avoid run-time errors.
As mentioned in section III-B, the implemented function `stima(a::mesh,b::element)` takes a `mesh` and an `element`. The type `element<:Tuple` itself is a user-defined subtype of `Tuple` and implemented as

```
element = Tuple{Vararg{Int}}.
```

The tuple is a build-in data structure with fixed-length where the number of entries are set by initialization. Another feature of tuples in Julia is that each entry can have its own type. They are constructed by using parentheses () where the entries are separated by commas:

```
X=(1,2,6.0)::Tuple{Int,Int,Float}.
```

It is important to notice, that the specific type of a tuple depends on the number and types of it's entries. The `Vararg` parameter states, that the amount of entries in an element can vary. Therefore the type of a linear LAGRANGE element with three DoF `NTuple{3,Int64}` is also a subtype of `element`.
To represent the triangulation in a single data-structure we defined the struct `mesh`, which is a collection of the three fields:

```
struct mesh
    coordinates::Array{Float64}
    elements::Array{element}
    boundaries::Array{element}
end.
```

In Julia a function can have multiple implementations, called methods, distinguished by the type annotations added to the arguments of the function. At run-time, a function call is dispatched to the most specific method applicable to the types of the arguments. To expand the functionality of our code, e.g. to allow the use of linear quadrilateral elements, it's sufficient to add a new method to the `stima()` function as shown below:

```
function stima(mesh,element::NTuple{3,Int64}) ...

function stima(mesh,element::NTuple{4,Int64}) ...
```

[ST]

B. Parallel computing

Our main goal with a parallel simulation is to achieve a lower overall run-time of the simulation. Some important ideas must be considered before trying to implement the parallelisation of a program. First question is whether the problem, or at least part of it, can be parallelised. That means if different tasks of the program can be independently calculated in different cores of the machine. The second important concept is the idea of the data transfer time used by the parallel approach. This is the time needed to move all the data to other cores so that the calculation can be done. Here we consider the total data transfer time, that means: the time to transfer the information to the cores and coming back from them. Also, the coordination time of this procedure is within the data transfer time. The third and last important point to be considered before trying the parallelisation is: how big is the problem or task to be parallelised. So that a positive "win time", i.e. the time difference between serial and parallel simulation ($T_{\text{win}} = T_{\text{ser}} - T_{\text{para}}$) is achieved, the problem must be big computationally speaking. The computational complexity, a measure of how many steps and/or how much time a given algorithm requires to solve a task in the worst case, is a good way to measure and define the computational size of a problem and say whether it is a big problem or not [11]. Due the great size of the problem, the "win time" gained by the multi-core calculation will be bigger than the data transfer time needed $T_{\text{win}} > T_{\text{trans}}$. The parallel simulation makes sense only if this last condition is satisfied otherwise the serial simulation will probably be faster [12].
After this first considerations we tried to implement the parallelisation in our code. To achieve that, we followed some crucial steps that allowed at the end the partial parallelisation of the algorithm:

- Tracking the possible tasks that can be parallelised
- Measurement of time of the tasks tracked in the first point
- Decision of which tasks could be parallelised in order to obtain the maximum "win time"
- Implementation of the parallel code in the most appropriate task decided in the third step
- Comparison of the serial and parallel simulation time

The second step of this list was achieved with the help of the macro `@time` of Julia. This macro executes the program indicated and print the time needed for the

action [13]. In this point only the execution time was a major comparison parameter for us, being the memory allocation not a decisive factor, at least in a first moment. To implement the steps described previously we took advantage of the parallel resources that the Julia PL offers to its users already in its core program [14]. We can divide them in three main categories:

- Coroutines (Green Threading)
- Multi-threads
- Multi-cores processing

The first and last ones were used in this work. Briefly explaining, the coroutines allow the communication and commands between serial and/or parallel tasks. The multi-cores allow the data transfer between the available cores and the calculation of these tasks in the cores. The multi-thread level was, as mentioned, not used in this work, since it is still a in test property of Julia. It is also worth to comment that Julia allows concurrency, which was also used in this work. Concurrency, differently from parallelism, is the calculation of the different tasks in just one core, therefore not at the same time and not parallel. However, these tasks can be divided and a partitioning and a better use of time of the serial calculations can be done so that the total time efficiency is increased.

During the first step of the above list aiming the parallelisation, we realize that the following tasks of our code could be parallelised: assembly of the global stiffness matrix; process to obtain the solution of the linear system of equations and the plotting of the results. Following the second step of out list, we measured the time of each of these tasks (2nd step of the list, see table I).

Task	Time [s]
Assembly of A	**3.59**
Solution of the linear system Eq.	1.95
Plotting of the Solution	4.99

TABLE I: Simulation time for different tasks (first version of the code): Mesh with 1365 vertices.

After that we decide, following the 3rd step of our list, to parallelise the assembling of the global stiffness matrix A because it seemed to be a more "natural" task to be paralyzed since it is done within a for loop with a large number of iterations and in which the elements of each calculation are independent of the elements of the prior iterations. The intuition would be that the assembling time of the matrix A is shorter than the plotting time and it could be better to parallelise this last task, however later on it was clear for us that with a bigger overall simulation time of our code, then the percentage of the time to assemble A in comparison with the total time also increases rapidly. In the section III-B one can see part of the implemented code showing the for loop used to calculate the elements of A. The independence of the elements of A to other elements of the same matrix is due to the choice of the Ansatz functions that are also independent of other points of the mesh.
To achieve the parallelisation of the loop presented in section III-B we used the Julia `@distributed` macro (4th step of the list). This macro allows a for loop to be parallelised. Basically, the specified range of the loop is partitioned and sent to other cores and locally executed in all workers available. If an optional reducer function is used, the macro `@distributed` does all the calculations in each core and at the final it performs a combination of the single results in a synchronous way. If this reduction function is not used, the macro executes every calculation asynchronously. That means it sends the tasks to all available workers and immediately after the completion of any task it sends the result of this task to be main core without waiting for the results of other tasks. As the calculation of the elements of the stiffness matrix are independently of each other, we chose the asynchronously implementation aiming a better performance in time. To perform all these calculations in different cores in a proper way, the `@distributed` uses the Julia coroutines, multi-core and concurrency properties. In the following code part one can see the for loop implemented in a parallel way with `@distributed`. No additional returner was used, therefore the iterations are done asynchronously and independently:

```
@time @distributed for i in 1:size(msh.elements,1)
A[collect(msh.elements[i]),
collect(msh.elements[i])]+=stima(msh,msh.elements[i])
                    end.
```

Finally, we reached the 5th step of our list to achieve the parallelism of our code, that is the comparison of the times of each serial and parallel computation (see fig. 4).

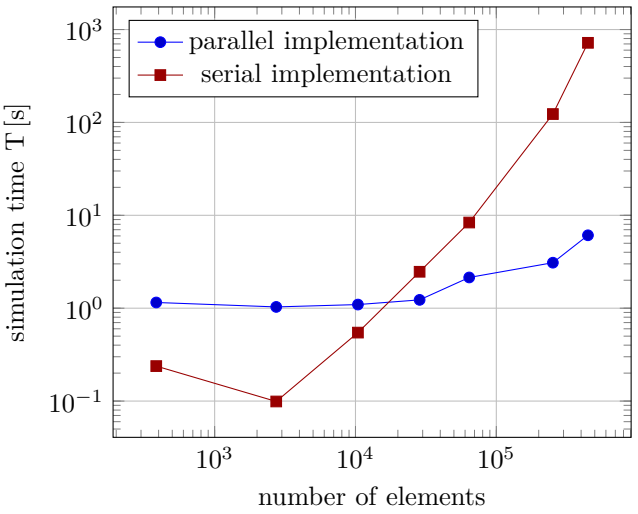

Fig. 4: Serial and parallel simulations run-time for different mesh resolutions: Calculated on a Intel Core i5 @ 4.0 GHz with 4 physical cores and 8 GB of system RAM.

From the figure 4 it is clear that for small meshes the parallel simulation has no advantages compared with the serial one since for the meshes with up to $\approx 10^4$ elements the serial simulation was faster. As explained before, for these size of meshes the relation $T_{\text{win}} - T_{\text{trans}} < 0$ indicates that the parallel simulation is slower since the "win time" can not compensate the amount of time to transfer the

data. However for simulations with a number of elements $> 2 \times 10^4$ the parallel simulation starts to have an outstanding performance in time in comparison with the serial one. It seems to be that the value $T_{\text{win}} - T_{\text{trans}}$ increases for bigger meshes, keeping the parallel simulation time almost constant for a number of elements $< 10^4$ and increasing slowly with a finer mesh. As the serial time increases in a logical way with a larger mesh, both simulation times become equal for a mesh size $\approx 2 \times 10^4$ elements and from this point on the parallel code has a visible advantage in run-time simulation.

It is therefore clear that even for this model treated in this paper a parallel simulation can deliver a great gain in time for a simulation. As explained before, for problems with even bigger tasks and calculations (e.g. larger number of elements) the tendency is that the parallel simulation delivers even better performances in time if one compares it with the serial simulation.

V. Convergence study

We test the accuracy of the finite element approximation of the solution of BVP $(\mathcal{B})$ by calculating an absolute nodal error for all N_e nodes within the mesh [6]. Choosing the Euclidean norm as vector norm for the resulting nodal error vector (N_e-dimensional) leads to the definition of a global nodal error δ:

$$\delta := \left(\sum_{i=1}^{N_e} \Big(\underbrace{\phi_k(x_i, y_i) - \phi(x_i, y_i)}_{\text{error at point } (x_i, y_i)} \Big)^2 \right)^{\frac{1}{2}} . \tag{7}$$

Therein is ϕ the analytic solution of BVP $(\mathcal{B})$, which can be expressed using a Cartesian coordinate chart [1]:

$$\phi : \mathbb{R}^2 \supset \overline{\Omega} \to \mathbb{R}, \quad (x, y) \mapsto \alpha_0 + \alpha_1 \ln\left((x^2 + y^2)^{\frac{1}{2}} \right) . \tag{8}$$

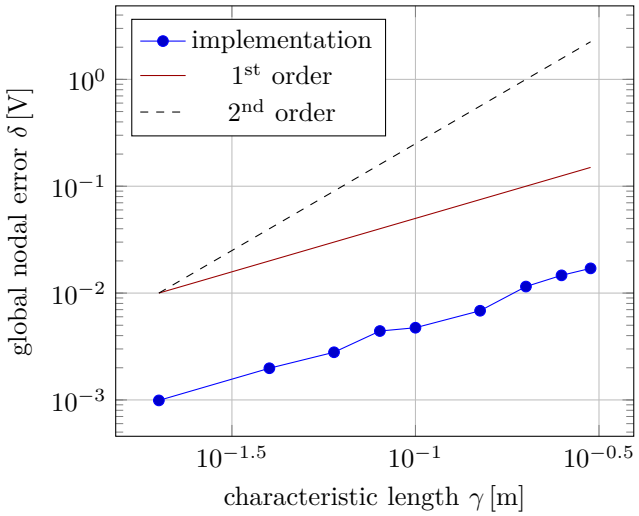

Fig. 5: Convergence study: The smaller γ, the finer the mesh.

Figure 5 summarizes the results of the convergence study: With error definition (7) ϕ_k converges linearly in γ^{-1} against ϕ. We want to emphasize, that the convergence order strongly depends on the choice of error validation, e.g. weighting δ with the respective N_e^{-1}, which can than be interpreted as a mean error per element, leads to the expected error of second order. [AP]

[1] $\alpha_0 := \frac{\phi_+ - \phi_-}{\ln(\frac{r_2}{r_1})}, \alpha_1 := \alpha_0 \ln(r_1)\phi_-$.

VI. Conclusion & Outlook

We used the Julia PL for a model problem in the context of computational electromagnetics and noticed some language specific advantages: The usage of type definitions led to a type stable, fast and readable code. Julia's ability for parallel computing increased heavily the performance of the finite element method implementation.

In future investigations it could be worthwhile to increase the polynomial order of the Ansatz functions to achieve a higher rate of convergence. We assume that this additional implementation effort is manageable, because of the variable definition of type `element` and the possibility of a multiple-dispatch for the assembly function `stima`. Increasing the polynomial order of the Ansatz functions would also lead to a larger number of DoF, where we aspect that the advantages of a parallel implementation are even more underlined. Further the computational domain could be expanded to three spatial dimensions, which could also increase the number of DoF. [AP]

References

[1] Bezanson, J., Edelman, A., Karpinski, S. and Shah, V., "Julia: A Fresh Approach to Numerical Computing", SIAM Review, issue 1, vol. 59, pp. 65–98, July 2015.
[2] Boffi, D and Brezzi, F and Fortin, M., "Mixed Finite Element Methods and Applications", Springer Science & Business Media, Berlin Heidelberg, 2013.
[3] Geuzaine, C. and Remacle, J., "Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities", International Journal for Numerical Methods in Engineering, vol. 79, no. 11, pp. 1309—1331, May 2009.
[4] Alberty, J., Carstensen, C. and Funken, S, "Remarks around 50 lines of Matlab: short finite element implementation", Numerical Algorithms, vol. 20, no. 2, pp. 117–137, August 1999.
[5] Schöps, S., De Gersem, H., Kurz, S. and Gangl, P., "Lecture notes on summer school PASIROM: Mathematische Modellierung und numerische Simulation von elektrischen Maschinen", Graduate School CE and TU Darmstadt, September 2018.
[6] Schuhmann, R., "Lecture: Finite Elemente in der Feldsimulation", TU Berlin, January 2019.
[7] Piwonski, A., Belz, D., Rezende, R. and Tchoumi, S., "GitLab Repository: wissenschaftliches_rechnen", https://gitlab.tubit.tu-berlin.de/damianb/wissenschaftliches_rechnen.git, TU Berlin, February 2019.
[8] Bezanson, J., Chen, J., Chung, B., Karpinski, S., Shah, V., Zoubritzky, L. and Vitek, J., "Julia: Dynamism and Performance Reconciled by Design", Proc. ACM Program. Lang., vol. 2, no. 120, pp. 1–23, November 2018.
[9] Pierce, B., "Types and Programming Languages", MIT Press, February 2002.
[10] Julia 1.1 Documentation, "Types", https://docs.julialang.org/en/v1/manual/types/index.html, visited in February 2019.
[11] Hall, L., "Computational Complexity", The Johns Hopkins University, http://www.esi2.us.es/~mbilbao/complexi.htm, visited in February 2019.
[12] García-Heredia, D., "A brief introduction to parallel computing in Julia", Universidad Carlos III de Madrid, June 2018.
[13] Julia 1.1 Documentation, "Essentials", https://docs.julialang.org/en/v1/base/base/#Base.@time, visited in February 2019.
[14] Julia 1.1 Documentation, "Parallel-Computing", https://docs.julialang.org/en/v1/manual/parallel-computing/index.html, visited in February 2019.