Rakenteiden Mekaniikka (Journal of Structural Mechanics)
Vol. 52, No 3, 2019, pp. 148–159
https://rakenteidenmekaniikka.journal.fi/index
https://doi.org/10.23998/rm.75103

VERTAISARVIOITU
KOLLEGIALT GRANSKAD
PEER-REVIEWED
www.tsv.fi/tunnus

Introduction to JuliaFEM an open-source FEM solver

Jukka Aho[1], Antti-Jussi Vuotikka and Tero Frondelius

Summary. This article briefly describes a new programming language Julia and a new innovative Finite Element Method (FEM) solver JuliaFEM. We selected an easy to understand example of a linear elasticity problem as a method for this introduction. We go through the example step by step and provide a detailed explanation of the different phases of the solution steps. The main result presented here demonstrates the scripting possibilities of JuliaFEM, both pre- and post-processing.

Key words: JuliaFEM, linear elasticity, finite element method

Received 18 September 2019. Accepted 22 March 2019. Published online 12 September 2019.

Introduction

JuliaFEM [1–3] is a new finite element method solver programmed using Julia. It evolved from the research and development conducted by Wärtsilä. The review article by Frondelius et al. [4] provides a historical overview of the research published by Wärtsilä in the field of structural analysis and dynamics. In fact, the very first published findings are related to FEM development.

At the moment, all traditional continuum elements with linear and quadratic Lagrange basis has been implemented. Moreover, JuliaFEM contains elements for the Euler-Bernoulli beam and Reissner-Mindlin plate bending problems. In JuliaFEM, physics is described with the help of Problem, and currently, we can simulate linear and nonlinear deformation of bodies given Dirichlet and Neumann boundary conditions and solve temperature of the body given thermal boundary conditions. Mathematically, governing equations are Cauchy momentum equation

$$\boldsymbol{\nabla} \cdot \boldsymbol{\sigma} - \boldsymbol{f} = \rho \ddot{\boldsymbol{u}}, \tag{1}$$

and the heat equation

$$\rho c_p \frac{\partial T}{\partial t} - \nabla \cdot (k \nabla T) = \dot{q}_v. \tag{2}$$

Or course the corresponding weak forms must be derived in order to calculate the local matrices for each element of the model. Based on the linearity of integration, global

[1]Corresponding author. ahojukka5@gmail.com

matrices of the discretized system are obtained from local element matrices by a finite element assembly procedure like described by Ibrahimbegovic [5]. According to Benzi [6], under suitable partitioning, any linear system

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{3}$$

can be cast in the form

$$\begin{bmatrix} \boldsymbol{K} & \boldsymbol{C}_1 \\ \boldsymbol{C}_2 & \boldsymbol{D} \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{bmatrix}. \tag{4}$$

In this sense, problems in JuliaFEM are divided to field problems and boundary problems, where the field problems typically populate $\boldsymbol{K}$, which results from the stiffness of the underlying discretized boundary value problem, and boundary problems are filling the rest of the matrices. The resulting system is, in principle, unwanted saddle point problem which turns out to be somewhat hard to solve, but in most of the cases, the unknown $\boldsymbol{\lambda}$ is possible to condensate out from the system to achieve a positive definite system. This setup gives more freedom to define problems containing more complicated couplings between domains and is a very natural starting point for formulating e.g. contact problems, where all four matrices and two vectors are used when discretizing the contact interface.

Once the global matrices are assembled, the next obvious question is what to do with them. In JuliaFEM, a concept called Analysis is defining the further operation which should be performed to the linear system. Currently, JuliaFEM supports quasistatic and dynamical analysis, natural frequency analysis and model reduction using Craig-Bampton method [2, 3].

By using programming techniques like templating and multiple dispatch, a developer can develop a new feature *independently* from the rest of the JuliaFEM code and finally deploy the functionality to the core code, when ready for production use. This kind of approach allows for an extremely modular program structure as the new features can be implemented on their own GitHub projects, see e.g. ModelReduction.jl [2], and finally, they can be imported to the meta-package JuliaFEM.jl with minimal changes to the old code, probably only having one import line like using ModelReduction.

Each new package, defining new features to JuliaFEM, of course, needs some common functionalities, which are provided by a minimal base package FEMBase.jl with high abstraction. As a result, JuliaFEM organization contains a lot of smaller packages, hopefully as a result of some academic research done in universities, which are then imported to the main package to extend the functionalities of JuliaFEM. In addition to JuliaFEM being extremely modular, the structure is also transparent and fair for software developers and academic researchers, giving all honor and credits of contributions to the corresponding authors.

It should also be mentioned, that new packages, which are extending the functionalities of JuliaFEM, are not restricted to implement new physics only. The developers of JuliaFEM are constantly working to build a framework, where all imaginable new features can be developed in their own packages and then deployed into a meta-package after they are ready for public beta testing or production. Such features may contain, for example, implementing new material models, parsers to read new mesh formats, writers to write the results of some analysis to hard drive or network storage, postprocessors, new analysis types and so on. Every feature can be implemented as a separate package. Described modular framework allows the researcher to work only with own, possibly private repository, and publish the code after a corresponding research paper is submitted to review.

Of course, the researcher is not obliged to publish the program code if he or she is doing research in a private company or keeping the code private is justified for further research.

There are basically two major advantages over existing open source FEM software. The first advantage is already mentioned programming technique called multiple dispatch, which in practice allows writing several functions with the same name and different types of arguments. This opens a possibility to create a modular design, where each package can define new types and implement their own function to e.g. assemble finite elements. Multiple dispatch is demonstrated with a simple example later in this article on page 151.

Another clear advantage is that JuliaFEM is a single programming language solution. There indeed exists several open source FEM software which uses, at least in some level, Python to make program code syntax appealing to research community and quick prototyping. However, the problems start very fast when it comes to scaling the code to solve bigger models which are simulated by industrial users. Python, to put it simply, is not fast enough, and its slowness is emphasized in operations with so-called hot loops, where some computationally inexpensive operation is performed multiple times. This scenario is demonstrated numerically on page 150 of this article. There are several workarounds to overcome this fact.

One popular workaround is to use two different programming languages, like C++ and Python, and automatically generate wrapper interface between programming languages using e.g. SWIG, like is done in FENiCS. While the approach basically works, there always exists an interface and while some features can be implemented using high-level Python, finally user must go into C++ side to implement features which utilizes hot loops, like evaluating material models in integration points. Another workaround is to try to vectorize operations in order to avoid hot loops, but it easily leads to an unconventional code structure, and finally, it's not possible to vectorize everything. These issues are already solved in JuliaFEM with a good choice of programming language, which is Julia.

Julia is a dynamic programming language [7], whose version 1.0 was released recently. One might ask, do we need yet another programming language? The key points that distinguish Julia from Python [8] are: a) Julia is just-in-time compiled [9], b) Julia uses multiple dispatch [10], and c) Julia is designed for scientific computing [11].

The first point means that any program code is first compiled and optimized to a low-level virtual machine code (LLVM [12]), and Julia can run as fast as native C. This feature is very advantageous when the program code contains very fast iterations running millions of times. In Python, the loops tend to run slowly, and if vectorization cannot be used in some part of the code, the only option to make the code run fast is to use some other programming language such as C/C++/Fortran to that part of the code.

A simple programming task is used for comparing Python and Julia. The task is to calculate a quadratic form $W = \langle \boldsymbol{u}, \boldsymbol{K}\boldsymbol{u} \rangle$, $N = 10^6$ times. This kind of task could be used to, for example, evaluate the potential energy of some structure, given its stiffness and deformation. Similar programming structures are inevitably required in software design, especially in the design of finite element solvers. The vectorized versions of Python and Julia are given in listings 1 and 2, and non-vectorized versions of the code are given in listings 3 and 4.

It is worth noting that because Julia is designed for scientific computing, matrices and vectors have highly optimized standard structures similar to that in programming languages such as MATLAB; therefore, there is no need to use external libraries as numpy [13] in Python. It results in a clear and understandable syntax.

Apart from a clear syntax for scientific computing, one of the more critical features is

```python
import numpy as np

def mult_1(N):
    W = 0.0
    K = np.matrix([[1.0, -1.0],
                   [-1.0,  1.0]])
    u = np.matrix([[1.0],
                   [2.0]])
    for n in range(N):
        W += (u.T * K * u)[0,0]
    return W
```

Listing 1: Python code, vectorized.

```julia
function mult_1(N)
    W = 0.0
    K = [ 1.0 -1.0
         -1.0  1.0]
    u = [1.0, 2.0]
    for n in 1:N
        W += u' * K * u
    end
    return W
end
```

Listing 2: Julia code, vectorized.

```python
def mult_2(N):
    W = 0.0
    K = np.matrix([[1.0, -1.0],
                   [-1.0,  1.0]])
    u = np.matrix([[1.0],
                   [2.0]])
    for n in range(N):
        for i in range(2):
            for j in range(2):
                W += u[i]*K[i,j]*u[j]
    return W
```

Listing 3: Python code, non-vectorized.

```julia
function mult_2(N)
    W = 0.0
    K = [ 1.0 -1.0
         -1.0  1.0]
    u = [1.0, 2.0]
    for n in 1:N
        for i in 1:2
            for j in 1:2
                W += u[i]*K[i,j]*u[j]
            end
        end
    end
    return W
end
```

Listing 4: Julia code, non-vectorized.

performance. The slowest code is non-vectorized Python code, given in listing 3. Compared to that, vectorized Python code, given in listing 1 runs about 8 times faster. However, Julia's performance is in its own class: the vectorized version of the code, given in listing 2, is 627 times faster than non-vectorized Python code. The fastest code is the non-vectorized Julia code, given in listing 4, which runs 2464 times faster than the slowest code. Noteworthy is that while the vectorized version of Python code is faster than non-vectorized, in Julia the situation is opposite. A non-vectorized version of Julia code runs faster than vectorized one because there are no memory allocations, which in general reduce performance significantly, in loops.

Another essential feature of Julia is multiple dispatch [10]. In Julia, the functions are generic, and they can have the same name with different input arguments. At the time of code execution, multiple dispatch ensures that the correct or the most appropriate function is called. This feature is demonstrated in listing 5: we define two structures, one for a 2-node linear line element and another one for a 3-node quadratic line element. We define a single function assemble!, and when this function is called, it returns the corresponding stiffness matrix depending on the input arguments. That is, for Seg2-element, the first function is called and for Seg3 element, the second function is called.

The advantage of multiple dispatch is that it is possible to define a modular software,

importing functionality from several submodules, where each submodule adds functionality to functions with the same name. That is, the same assemble!-function can be used to assemble different kinds of elements with different physics, and the variations of the functions can be defined in different modules.

```
struct Seg2
    Ke :: Matrix{Float64}
end

struct Seg3
    Ke :: Matrix{Float64}
end

Seg2() = Seg2(zeros(2,2))
Seg3() = Seg3(zeros(3,3))

function assemble!(element::Seg2)
    @info("Assembling 2-node linear line element.")
    element.Ke .= [1.0 -1.0; -1.0 1.0]
    return nothing
end

function assemble!(element::Seg3)
    @info("Assembling 3-node quadratic line element.")
    element.Ke .= 1.0/6.0 * [7.0 1.0 -8.0; 1.0 7.0 -8.0; -8.0 -8.0 16.0]
    return nothing
end

element1 = Seg2()
element2 = Seg3()
assemble!(element1)
assemble!(element2)

# output

Info: Assembling 2-node linear line element.
Info: Assembling 3-node quadratic line element.
```

Listing 5: Example of using multiple dispatch to assemble two different elements. Julia finds a suitable function for the given input arguments. The first function call for assemble! reaches the first function body and the second function call reaches the second function body.

One question that a reader might have on Julia is why with all these great features there are not more Julia codes popping up? Probably the simplest explanation is that Julia was found in 2012 and it is a relatively new programming language. Julia programming language gets its first stable release during JuliaCon 2018, which was kept on London in August of 2018. After the launch of 1.0 version, according to GitHub statistics, the interest to the programming language skyrocketed, which hints that large masses of potential users and developers was waiting for the first stable release. A number of active developers is

still a small fraction compared to the Python, but the community is growing rapidly since the release of 1.0 version, as shown in figure 1.

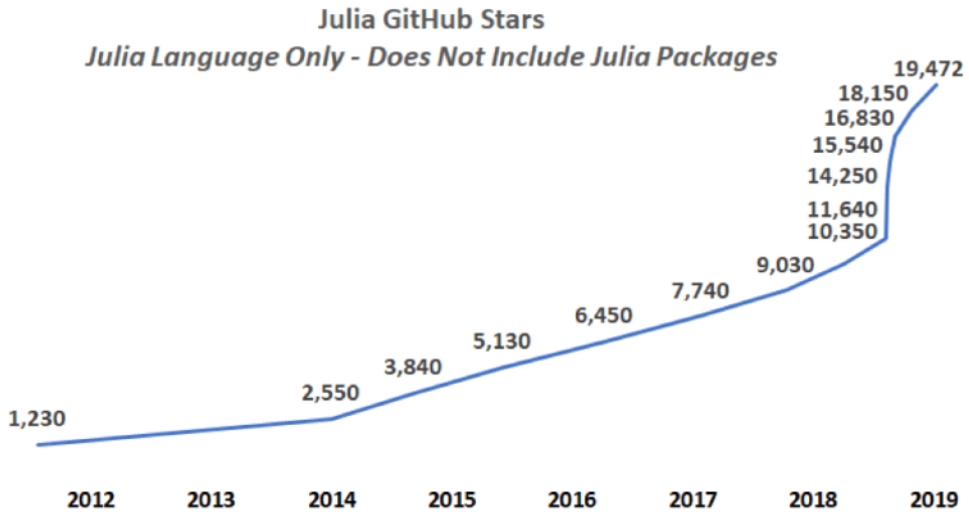


Figure 1: The number of GitHub Stars for Julia has also doubled over the past year. Image: Julia Computing.

With this introduction to Julia, we next introduce JuliaFEM, a new finite element solver, written purely using Julia. The JuliaFEM project develops open-source software for a reliable, scalable, and distributed Finite Element Method. The JuliaFEM software library is a framework that allows for distributed processing of massive Finite Element Models across clusters of computers using simple programming models [1]. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. The basic design principle is: everything is nonlinear. All physics models are nonlinear, from which linearization is made as a special case.

On the one hand, the vision of JuliaFEM includes the capability for massive parallelization using multiple computers with MPI and threading as well as cloud computing resources using Amazon, Azure, and Google Cloud services together with an internal company server [1]. On the other hand, the real application complexity includes the simulation model complexity as well as geometric complexity. The reuse of the existing material models, as well as the entire simulation model, is considered a crucial feature of the JuliaFEM package.

Recreating the wheel again is definitely not anybody's goal, and therefore we try to use and embrace good practices and formats as much as possible. We have implemented Abaqus / CalculiX input-file format support, and in future, we may extend support to other FEM solver formats. The use of modern development environments enables users to achieve fast development time and high productivity. For developing and creating new ideas and tutorials, we have used Jupyter notebooks [14], which helps in making easy-to-use handouts.

The user interface for JuliaFEM is Jupyter Notebook, and Julia language itself is a real programming language. It makes it possible to use JuliaFEM as a part of a bigger solution cycle, including, for example, data mining, automatic geometry modifications, mesh generation, solution, and post-processing, and to enable efficient optimization loops. [1]

Numerical example

A detailed workflow for solving linear elasticity is given. In continuum mechanics [15], the balance of the linear momentum for a continuous medium, which occupies domain Ω, may be specified as $\boldsymbol{\nabla} \cdot \boldsymbol{\sigma} - \boldsymbol{f} = \rho \ddot{\boldsymbol{u}}$, where $\boldsymbol{\sigma}$ is the Cauchy stress tensor and $\boldsymbol{f}$ is the applied body force per unit volume. Due to the balance of angular momentum, Cauchy stress tensor is symmetric, that is, $\boldsymbol{\sigma} = \boldsymbol{\sigma}^{\mathrm{T}}$. The inertial terms are neglected in this example, so the equilibrium equation considered to solve is $\boldsymbol{\nabla} \cdot \boldsymbol{\sigma} - \boldsymbol{f} = \boldsymbol{0}$. The boundary conditions are defined by dividing the boundary $\partial\Omega$ of the calculation domain Ω into two disjoint subsets $\partial\Omega = \Gamma_\sigma \cup \Gamma_\mathrm{u}$, $\Gamma_\sigma \cap \Gamma_\mathrm{u} = \emptyset$, and requiring that $\boldsymbol{u} = \hat{\boldsymbol{u}}$ is known at Γ_u and $\boldsymbol{\sigma} \cdot \boldsymbol{n} = \hat{\boldsymbol{t}}$ is known at Γ_σ. Here, $\boldsymbol{\sigma} = \mathcal{C} : \boldsymbol{\varepsilon}$, where $\mathcal{C}$ is the fourth-order elasticity tensor and $\boldsymbol{\varepsilon}$ is a small strain tensor. The weak form is obtained by multiplying the linear momentum equation by a test function and using the divergence theorem. An example model is shown in figure 2. The geometry is modeled and meshed using Salome, which is an open-source CAD software ready for the open-source toolchain Salome + JuliaFEM [1] + Paraview [16]. The mesh is shown in Figure 3a, and the model properties are tabulated in Table 3b.

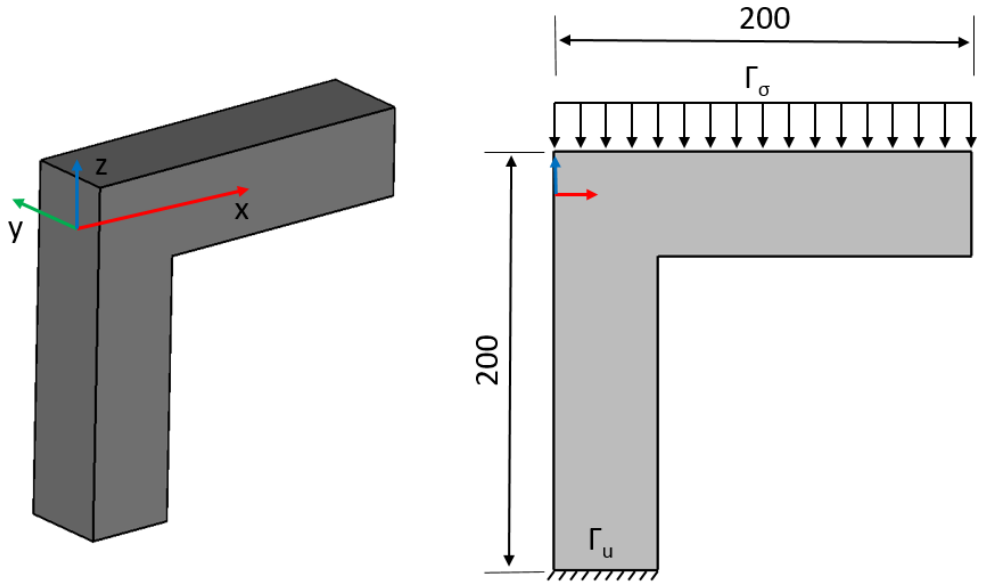
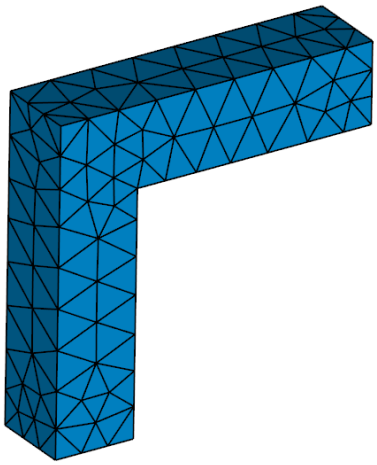


Figure 2: Calculation model. The geometry is modeled using Salome platform, an open-source CAD software. The thickness of the calculation domain is 50.

The typical workflow for using JuliaFEM to solve partial differential equations is as follows: 1) read mesh, 2) create elements, 3) update element properties, 4) create problems, 5) create analysis, and 6) run analysis. Now, we look at these steps in detail.

The first thing to do is to define the computation domain where the numerical simulation is performed. In JuliaFEM, this is usually done by reading an already discretized geometry, also known as finite element mesh, from a computer's hard disk. Currently, JuliaFEM supports reading a mesh from Code Aster [17] file format (using `aster_read_mesh`) and from ABAQUS file format (using `abaqus_read_mesh`). The next step is to create one or several sets of elements from a mesh. This is done using a function `create_elements(mesh, set_name)`. These steps are shown in listing 6.



(a) Finite element mesh.

Property	Value
Youngs modulus	200.0e6
Poissons ratio	0.3
Surface pressure	100.0e3

(b) Model properties

Figure 3: Finite element mesh and model properties. The solid model is meshed using Salome platform. Mesh contains 538 linear tetrahedron elements (Tet4).

```
using JuliaFEM

mesh = aster_read_mesh("mesh.med")
body_elements = create_elements(mesh, "body")
traction_elements = create_elements(mesh, "traction")
bc_elements = create_elements(mesh, "bc")
```

Listing 6: Preprocessing steps in JuliaFEM: read mesh from a file and create elements based on the mesh.

In JuliaFEM, all the properties of elements are given using fields. Fields may depend on spatial coordinate or time. The fields of elements are updated using update!-function. Listing 7 shows this step.

```
update!(body_elements, "youngs modulus", 200.0e6)
update!(body_elements, "poissons ratio", 0.3)
update!(traction_elements, "surface pressure", 100.0e3)
for i in 1:3
    update!(bc_elements, "displacement $i", 0.0)
end
```

Listing 7: Update elements with fields. Everything is defined as fields, and all the fields can depend on time, spatial location, or other fields.

Like explained in introduction, the physics that is considered for the solution is given by using Problem. The problems are defined by giving the problem type as the first argument, the problem name as the second argument, and the dimension of the problem (which indicates the degrees of freedom connected to each node) as the last argument. In this case, the problem type is Elasticity . We also need to define additional boundary

problem type Dirichlet to handle Dirichlet boundary condition. After the problems are created, the elements are added to them by using the function add_elements!; see listing 8.

```
13 body = Problem(Elasticity, "body", 3)
14 traction = Problem(Elasticity, "traction", 3)
15 bc = Problem(Dirichlet, "bc", 3, "displacement")
16 add_elements!(body, body_elements)
17 add_elements!(traction, traction_elements)
18 add_elements!(bc, bc_elements)
```

Listing 8: Define new problems and add elements to the problems. Problem defines the physics considered for the solution.

After the simulation domain and physics are defined, the type of analysis is defined. For simplicity, a linear quasistatic analysis of the given problem is performed. The problems are added to the analysis using add_problems!. The final step is to request the results of the analysis to be written to the disk for later use and perform the analysis. Currently, Xdmf [18] output is supported, which can be read using ParaView. These steps are shown in listing 9. The deformed shape is shown in figure 4.

```
19 analysis = Analysis(Linear, "step 0")
20 add_problems!(analysis, body, traction, bc)
21 xdmf = Xdmf("results"; overwrite=true)
22 add_results_writer!(analysis, xdmf)
23 run!(analysis)
24 close(xdmf)
```

Listing 9: Define new analysis, add problems to the analysis. Define results writer and run the analysis.

After the analysis is ready, the types and variables can be accessed using REPL or Jupyter notebook for further postprocessing. The simulation can also be written into a function, to be a part of a more extensive analysis process. For example, it is possible to request a deflection line of the body along the x axis, which is shown in listing 10. The definition of the axis coordinate system is shown in figure 2.

```
25 using Plots
26 x = range(0.0, stop=200.0, length=100)
27 u3(x) = body("displacement", [x, 0.0, 0.0], 0.0)[3]
28 plot(x, u3, title="Displacement in z-direction along x-axis", color=:black)
```

Listing 10: Postprocessing in programming environment: interpolate displacement in the z direction along the x axis.

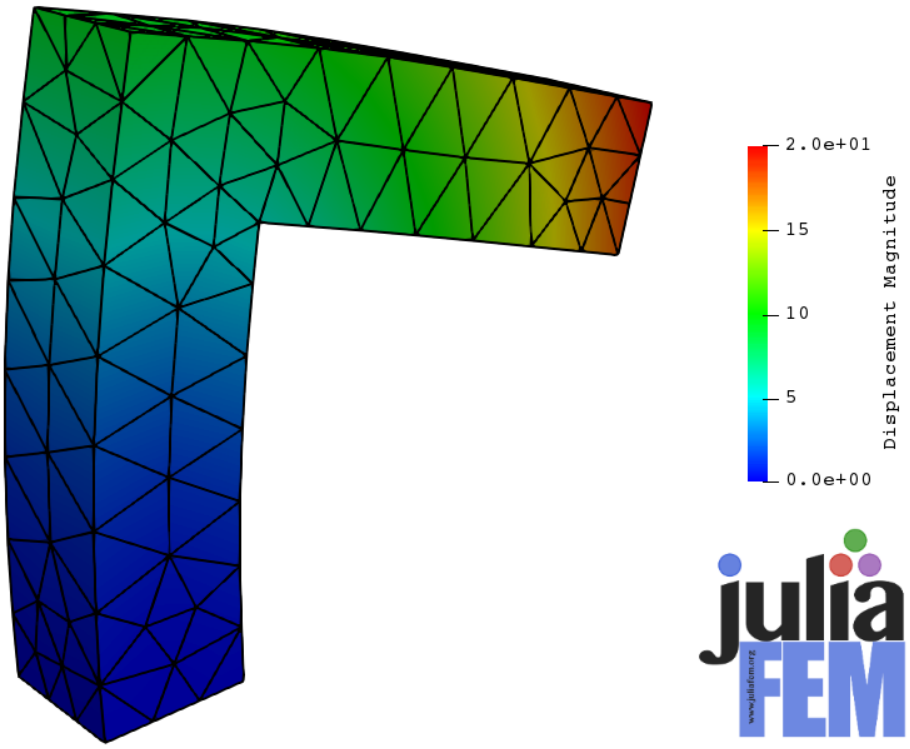


Figure 4: Results, deformed shape. The results are read from the Xdmf file format and visualized using ParaView.

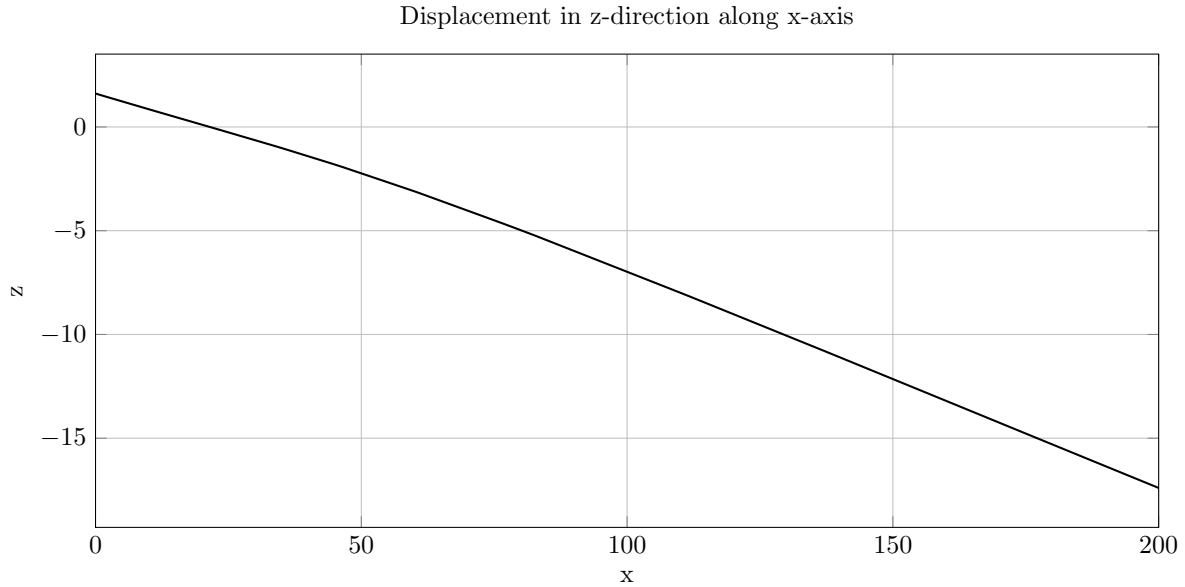


Figure 5: Displacement is interpolated inside domain Ω in the z direction along the x axis.

Conclusions

This article described the basic usage of JuliaFEM. A brief introduction to Julia programming language was given. The use of JuliaFEM for solving the deformation of a body subjected to surface pressure was demonstrated. As a programming language, Julia is highly suitable for developing a finite element method solver due to its excellent performance. The clean syntax of Julia and the careful user interface design of JuliaFEM makes it possible to write simple Julia script files for performing typical FEM analyses. Because the modeling is done inside the scripting environment, very powerful postprocessing and optimization tasks could be performed.

References

[1] Tero Frondelius and Jukka Aho. JuliaFEM - open source solver for both industrial and academia usage. *Rakenteiden Mekaniikka*, 50(3):229–233, 2017. URL: https://doi.org/10.23998/rm.64224.

[2] Marja Rapo, Jukka Aho, Hannu Koivurova, and Tero Frondelius. Implementing model reduction to the JuliaFEM platform. *Rakenteiden Mekaniikka*, 51(1):36–54, 2018. URL: https://doi.org/10.23998/rm.69026.

[3] Marja Rapo, Jukka Aho, and Tero Frondelius. Natural frequency calculations with JuliaFEM. *Rakenteiden Mekaniikka*, 50(3):300–303, 2017. URL: https://doi.org/10.23998/rm.65040.

[4] Tero Frondelius, Hannu Tienhaara, and Mauri Haataja. History of structural analysis & dynamics of Wärtsilä medium speed engines. *Rakenteiden Mekaniikka*, 51(2):1–31, 2018. URL: https://doi.org/10.23998/rm.69735.

[5] Adnan Ibrahimbegovic. *Nonlinear solid mechanics*. Springer, New York, 2009.

[6] Michele Benzi, Gene H. Golub, and Jörg Liesen. Numerical solution of saddle point problems. *Acta Numerica*, 14(1):1–137, may 2005. doi:10.1017/S0962492904000212.

[7] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. URL: https://doi.org/10.1137/141000671.

[8] Guido van Rossum and Jelke de Boer. Linking a stub generator (ail) to a prototyping language (python). In *Proceedings of the Spring 1991 EurOpen Conference, Troms, Norway*, pages 229–247, 1991.

[9] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[10] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. In *Acm sigplan notices*, volume 43, pages 563–582. ACM, 2008.

[11] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[12] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

[13] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[14] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.

[15] Gerhard A Holzapfel. Nonlinear solid mechanics: a continuum approach for engineering science. *Meccanica*, 37(4):489–490, 2002.

[16] Utkarsh Ayachit. *The paraview guide: a parallel visualization application*. Kitware, Inc., 2015.

[17] User Manual. Booklet u1. 0-: Introduction to code aster. *Document: UT02. 00*.

[18] E Mark et al. Enhancements to the extensible data model and format (xdmf). In *DoD High Performance Computing Modernization Program Users Group Conference, 2007*, pages 322–327. IEEE, 2007.

Jukka Aho
ahojukka5@gmail.com

Antti-Jusssi Vuotikka
Global Boiler Works Oy
Lumijoentie 8
90400 Oulu
antti-jussi.vuotikka@gbw.fi

Tero Frondelius
Wärtsilä
Järvikatu 2-4
65100 Vaasa
tero.frondelius@wartsila.com

Tero Frondelius
Oulu University
Pentti Kaiteran katu 1
90014 Oulu
tero.frondelius@oulu.fi