Implementation of finite element method for solid mechanics problems in Julia programming language

A D Antoshkin, A V Cherednichenko and I Yu Savelyeva
Department of Applied Mathematics, Bauman Moscow State University, 5/1, Baumanskaya 2-ya, Moscow, 105005, Russia
E-mail: antoshkin.alexandr@gmail.com

Abstract. The paper proposes finite element method algorithm implemented in Julia programming language. The main features of Julia programming language are described. The results of Kirsch problem solution are presented. Comparing obtained results with appropriate calculations with open-source finite element software Code_Aster are also presented.

Key words: finite element method, elasticity theory, Kirsch problem, Julia programming language, Code_Aster.

1. Introduction
In present time finite element method is one of the most popular ways to simulate different physical phenomena. Particularly we can use it for the analysis of structures, fluids and solids, simulating heat transfer models and other problems.

The main purpose of this work is to implement finite element method algorithm in Julia programming language. Provided algorithm should be easy-scalable. To achieve that we use some Julia programming language features to provide such architecture of application [1]. We simulate Kirsch problem to test implemented algorithm [2].

2. Julia programming language
Julia programming language is the high-level dynamic programming language with JIT compiler which allows to translate bytecode to machine one at run time. The main programming paradigm is multiple dispatch. The language is mostly intended for implementation of algorithms with complex math computations. Despite on that it also can be used for more common problems.

Julia has a lot of useful tools for different mathematical problems [3]. One of the most commonly used tools in this work is built-in module "LinearAlgebra" which implements a lot of algorithms for solving different linear algebra problems. Programmer can use supported operations such as "tr" (matrix trace), "det" (matrix determinant), "eigvals" (eigenvalues), "eigvecs" (eigenvectors), etc. Also Julia supports different special matrices. To use such matrix programmer just needs to call appropriate function-constructor (e.g. "Symmetric()" or "Diagonal()"). Further interaction with these matrices is similar to usual ones. Thus programmer can use default operations and Julia will choose the best method to perform them according to the type of given matrix.

With more simple syntax most dynamic programming languages has low performance compared to compiled programming languages such as C or C++. To improve performance
Julia uses dynamic compiler based on LLVM (Low Level Virtual Machine) toolchain. This leads to significant performance improvement in Julia.

One of Julia features that we use in this work to achieve scalability of application is "Modules". Modules in Julia are represented as separated scopes of variables. Programmer implements different modules independently. Then it can be used via import using "import" or "using" keyword.

Another feature which appears to be very useful in this work is "Abstract Types". With this feature we can define abstract type and "inherit" subtypes from it. It’s not so flexible as inherit mechanism in some OOP languages (in fact these mechanisms are completely different) but in this paper we can use it to achieve significant flexibility of application algorithm.

3. Application architecture
To provide easy-scalable architecture we divide application into 3 main blocks (figure 1):

- Interface module. This module describes mesh import, input of basic process parameters and export of obtained results. For all model parameters and conditions import we use simple text file edited in special format. To set main process parameters we use some predefined keywords (e.g. "PRatio" for Poisson’s ratio, "YoungMod" for Young’s modulus). In this work we use open-source data analysis and visualization application Paraview. One of the most convenient formats which can be read by Paraview is it’s own format "VTK". That is why we implement special function to convert obtained results to VTK format for the further analysis. In future in this module we can implement appropriate functions to convert results to other different formats.

- Mesh module. This module contains mesh processing, converting mesh to convenient format, export of converted mesh to processing core. Mesh is stored as a set of nodes coordinates and a set of elements which contains these nodes. Nodes coordinates as well as elements are stored as a vector of "Tuples":

```julia
mutable struct Mesh2D_T
    "Matrix of nodes (contains node’s coordinates)"
    nodes::Vector{Tuple{Vararg{Float64}}}
    "Matrix of elements (contains elements’s nodes)"
    elements::Vector{Tuple{Vararg{Int}}}
end # Mesh2D_T
```

The variables of such type have preset size and it can’t be explicitly changed after initialization. To distinguish different mesh types we define enumeration using macro "@enum":

```julia
@enum meshType begin
    Quad4Pts2D
    Quad8Pts2D
end
```

- Core module. This module describes calculations according to appropriate mesh and given process parameters (finite element method implementation). Classic finite element algorithm implementation consists of several steps: calculation of local stiffness matrices and local force vectors, assembling of global system of linear equations, applying of boundary conditions, solving of obtained system of linear equations. Material properties and possible boundary conditions we define as enumerations (assuming that material is linear isotropic):
To describe constants for different problems we also define appropriate enumerate:

@enum mechProblemType begin
  plainStrain
  plainStress
end

All process parameters during calculations we can store as appropriate structure:

mutable struct processPars
  materialProperties::Dict{materialProperty, Real}
  bc::Dict{Int, bc} # {NumberOfPoint, TypeOfBC}
  load::Dict{Array{Int, 1}, Array{Float64, 1}} # {NumberOfPoint, ForceVector}
  mesh::Mesh2D_T
end

To solve linear system we use appropriate operator "\" from built-in module "LinearAlgebra":

solution = globalK \ loadVector

In the code fragment above globalK is the global stiffness matrix, loadVector is the right part of system of equations. This operator will automatically choose the most suitable method to solve given system of equations. Method choice is based on matrix type. Thus for calculations speed-up we can set matrix in special forms. For instance we can set matrix as symmetric one using "Symmetric" module.

Figure 1. Architecture of application.

Each block as well as the expressions for the specific finite element type we implement as separate Julia module. Such using of Julia modules helps to simplify possible modifications of
application to solve different problems. Thus application modifications can be independent and
doesn’t impact on base blocks such as processing core and interface.

To simplify further usage of different types of finite elements we define an abstract type of
finite element. This type we consider as a header of global finite element types system:

abstract type FiniteElement end

Along with header type of finite element we define global enumerate of possible finite element
types:

@enum FETypes begin
    Quad4TypeID
    Quad8TypeID
end

Then we can implement specific type of finite element as separate Julia module. To add this
type to global types system it’s necessary to define subtype-indicator of "FiniteElement":

struct Quad4Type <: FiniteElement
    name::String
end

In fact this type is not used in the usual way. It means that we don’t make any operations on
variables of this type. Instead of this we use it as an indicator to allow Julia to choose function
for the specific type of finite element. Thus fields in struct above are not necessary but we store
there the name of finite element type for convenience. Here is an example of common function
for different types of finite element:

function ElementTypes.gradMatr(r, s, xCoords, yCoords, elemTypeInd::Quad4Type)

In code fragment above the last argument in function is used to define which instance of
appropriate function is assumed to calculate gradient matrix. This is a notable example of
multiple dispatch feature in Julia programming language.

For further calculations we implement displacement based finite element algorithm in
application core. To simplify computation process we consider isoparametric finite elements.

4. Finite element method
In this work we implement displacement based finite element method. Let us divide given body
to the discrete set of elements [4]. Consider nodes displacements vector:

$$\vec{U} = (u_{1x}, u_{1y}, u_{1z}, ..., u_{nx}, u_{ny}, u_{nz})^T,$$

where $u_{ix}$, $u_{iy}$, $u_{iz}$ are the displacements of $i$-th node along appropriate axes, $n$ is number of
nodes.

Then for element $e$ we can assume:

$$U_e(x,y,z) = H_e(x,y,z)\vec{U},$$
$$\varepsilon_e(x,y,z) = B_e(x,y,z)\vec{U},$$

where $U_e(x,y,z)$ and $\varepsilon_e(x,y,z)$ are displacement and deformation matrices for element $e$
appropriately.

Stresses in element $e$ can be expressed using the Hook’s law:

$$\sigma_e = C_e\varepsilon_e + \sigma_0^e,$$
where
\[ \sigma = (\sigma_{xx} \sigma_{yy} \sigma_{zz} \sigma_{xy} \sigma_{yz} \sigma_{zx})^T, \quad \varepsilon = (\varepsilon_{xx} \varepsilon_{yy} \varepsilon_{zz} \gamma_{xy} \gamma_{yz} \gamma_{zx})^T, \] (4)

\( C_e \) is the elasticity matrix corresponding to appropriate material, \( \sigma_e^0 \) is the initial stresses in element \( e \), \( \nu \) is the Poisson’s ratio and \( E \) is the Young’s modulus.

According to principle of virtual displacements we can assume:
\[
\tilde{U}^T \sum_e \left( \int_{V_e} B^T_e C_e B_e \, dV_e \right) U = \tilde{U}^T \sum_e \left( \int_{V_e} H^T_e F^e_V \, dV_e \right) + \tilde{U}^T \sum_e \left( \int_{S_e} H^T_{S,e} F^e_S \, dS_e \right) + \tilde{U}^T \tilde{F},
\] (5)

where \( \tilde{F} = \sum_i H^T_{c,i} F_{c,i} \), \( F_S \) is the vector of element surface forces, \( F_V \) is the vector of element body forces, \( F_C \) is the nodal concentrated loads.

For isoparametric finite element we can assume: \( dV_e = \det J \, dr \, ds \, dt \), where \( r, s, t \) are axes of natural coordinate system, \( J \) is the Jacobian matrix for transformation from initial coordinate system to the natural one. Assuming \( \tilde{U}^T = I \), where \( I \) is the identity matrix we can consider system of linear equations:
\[
K U = F,
\] (6)

\[
K = \sum_e K_e,
\] (7)

\[
K_e = \int_{V_e} B^T_e C_e B_e \, dV_e \det J \, dr \, ds \, dt,
\] (8)

\[
F = \sum_e \left( \int_{V_e} H^T_e F^e_V \, dV_e \right) + \sum_e \left( \int_{S_e} H^T_{S,e} F^e_S \, dS_e \right) + \tilde{F}.
\] (9)

Matrix \( K \) is called stiffness matrix of finite elements ensemble, \( K_e \) is the local stiffness matrix of element \( e \), \( F \) is load vector which takes into account the impact of element body forces, element surface forces, initial stresses and nodal concentrated loads.

To apply boundary conditions we can represent given system of equations as follows:
\[
\begin{pmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{pmatrix} \begin{pmatrix} U_1 \\ U_2 \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix},
\] (10)

where \( U_1 \) — unknown nodes displacements, \( U_2 \) — given boundary conditions. Thus to find unknown displacements we should solve system of equations:
\[
K_{11} U_1 = F_1 - K_{12} U_2.
\] (11)

Deformations and displacements are connected via compatibility condition [5].
5. Kirsch problem

For further testing of finite element method algorithm let us describe Kirsch problem. Consider square plate with hole inside (figure 2). Side length is \( l = 200 \text{ mm} \), radius of the hole is \( R = 10 \text{ mm} \).

Pressure \( P = 10 \text{ MPa} \) is applied to the left and right sides of given plate. Plate parameters: Young’s modulus: \( E = 200000 \text{ MPa} \), Poisson’s ratio: \( \nu = 0.3 \). The material of given plate is perfectly elastic. It is necessary to determine the displacements field, corresponding strains and stresses.

![Figure 2. Kirsch problem.](image)

To provide more simple numerical analysis we can consider appropriate quarter plate (figure 3). This model should be equivalent to previously described one. To achieve that we can set special boundary conditions: on the left side of given plate we fix displacements along x-axis \( (u|_{x=0} = 0) \) and on the bottom side we fix displacements along y-axis \( (v|_{y=0} = 0) \).

![Figure 3. Quarter plate.](image)
6. Results and discussion
Let us consider 4-nodes mesh which we can build in open-source pre- and post-processing software Salome (figure 4).

![4-nodes mesh](image)

**Figure 4.** 4-nodes mesh.

To verify obtained results we compare it with appropriate calculations in Code_Aster [6]. Code_Aster is free and open source finite element package for solving structural analysis problems [7].

For better analysis we should calculate value that is regardless of specific coordinate system. To provide this we can use von Mises stress. Von Mises stress distribution fields in case of implementation in Julia and in case of calculations via Code_Aster are shown in figure 5.

![Von Mises stress distribution](image)

**Figure 5.** Von Mises stress distribution (4-nodes finite element): a) Implementation in Julia; b) Code_Aster.
To provide numerical comparison of obtained results we submit the results in several plate points (table 1). Kirsch problem has known analytical solution based on Saint-Venant’s principle. As soon as according to that principle we can observe stresses concentration near the hole let us submit analytical solution only on the boundary of given hole.

**Table 1.** Von Mises stress comparison (4-nodes finite element), MPa.

| Code | Implementation in Julia | Relative error |
|------|-------------------------|----------------|
| $(x, y) = (100, 100)$ | 9.99931 | 0.00006 |
| $(x, y) = (100, 0)$ | 9.71864 | 0.00016 |
| $(x, y) = (0, 100)$ | 9.66293 | 0.00210 |
| $(x, y) = (0, 10)$ | 27.98610 | 0.01274 |
| $(x, y) = (10, 0)$ | 8.98065 | 0.14661 |

Now let’s consider the same problem with 8-nodes mesh (figure 6). To implement this case we need to make a module which should contain all necessary expressions for model with 8-nodes finite element.

**Figure 6.** 8-nodes mesh.

Comparison of von Mises stress distribution is given in figure 7. Numerical analysis of obtained results is given in table 2.
According to given results we can conclude that obtained distribution is correct. Notice that relative error is bigger near the hole where stresses are concentrated. This is due to different ways to interpolate deformation values to define stresses at mesh nodes.

### 7. Conclusion

In this work base blocks of finite element method application were implemented. Architecture which allows to add new processing blocks and modify existing ones to solve different engineering problems was built. Classic elasticity theory problem as an example was simulated. Results of applying displacement-based finite element method to described problem were obtained and analysed. A conclusion about given results correctness was made.

### Acknowledgments

The research was supported by the Ministry of Science and Higher Education of the Russian Federation, project No. 0705-2020-0047.
References

[1] Official website of julia language, URL: https://julialang.org/ (accessed 26.12.2020)
[2] Timoshenko S P and Goodier J N 1951 Theory of Elasticity (New York: McGraw-Hill Book Company)
[3] Bezanson J, Edelman A, Karpinski S and Shah V B 2015 Julia: a fresh approach to numerical computing (Preprint 1411.1607)
[4] Bathe K J 2014 Finite Element Procedures (Prentice Hall: Pearson Education)
[5] Mushelishvili N I 1966 Some basic elasticity theory problems (Moscow: Nauka)
[6] Code_aster documentation v12, URL: https://www.code-aster.org/V2/doc/v12/en/ (accessed 26.12.2020)
[7] Cherednichenko A V, Maksimova E A and Savelyeva I Y 2019 Material Science and Engineering 747