TEKNILLINEN TIEDEKUNTA

IMPLEMENTATION OF A 2D BEAM ELEMENT TO JULIAFEM

Ville Jämsä

PROGRAMME OF MECHANICAL ENGINEERING

Kandidaatintyö 2018

TIIVISTELMÄ

Implementation of a 2D beam element to JuliaFEM

Ville Jämsä

Oulun yliopisto, Konetekniikan tutkinto-ohjelma

Kandidaatintyö 2018, 21 s. + 4 liitettä

Työn ohjaaja(t) yliopistolla: Hannu Koivurova

Tämän työn tavoitteena oli implementoida kaksiulotteinen palkkielementti Julia-koodikielellä JuliaFEM:iin, joka on Julia-koodikielellä koodattu avoimen lähdekoodin elementtimenetelmäohjelmisto. Palkkielementin taustalla on Euler-Bernoullin palkkiteoria, joka sopii hyvin hoikille palkeille. Koodi integroi palkkielementin jäykkyysmatriisin, massamatriisin ja ekvilanttisten solmuvoimien vektorin Gaussin numeerisella integroimismenetelmällä. Työssä käydään läpi palkkiteoria ja yhtälöt palkkielementin matriisien taustalla. Gaussin numeerinen integroimismenetelmä esitellään lyhyesti.

Koodilla saadaan samat jäykkyysmatriisit, massamatriisit ja ekvivalenttisten solmuvoimien vektorit kuin käsinlaskennalla, minkä perusteella voidaan todeta koodin toimivan oikein. Jatkossa olisi tarkoitus implementoida koodi myös kolmiulotteiselle palkille.

Asiasanat: Beam element, Euler-Bernoulli, JuliaFEM

ABSTRACT

Aim for this thesis was to implement 2D beam element to JuliaFEM – an open source finite element method solver which is written in Julia code language. The theory behind the beam element is Euler-Bernoulli beam theory which works well in case of slender beams. The code integrates beam element's stiffness matrix, mass matrix and equivalent forces vector with Gaussian quadrature. In this thesis the beam theory and equations behind the matrices are introduced. Gaussian quadrature is introduced briefly.

The code gives the same matrices as calculations by hand and it can be stated that it works correctly. In future it is intended to implement beam element for also three dimensional cases.

Keywords: Beam element, Euler-Bernoulli, JuliaFEM

FOREWORD

The main goal of this work was to implement 2D beam element to JuliaFEM which is an open source finite element method solver. The work was done for Wärtsilä Finland Oy where I currently work as a summer trainee. I did the coding and writing during spring 2018 in Oulu.

By doing this thesis I got to learn how finite element method solvers work, study finite element methods deeper and learn basics of open source development. I developed many new skills in many areas especially coding and finite element method.

I would like to thank Tero Frondelius for granting this opportunity to me. A big thanks goes to Jukka Aho and Marja Rapo for all their invaluable help in coding. I would also like to thank my bachelor's thesis supervisor Hannu Koivurova from University of Oulu for his help, advice and support. Last but not least I would like to thank my family, girlfriend and roommate for all their love and support.

Ville Jämsä

Ville Jämsä

TABLE OF CONTENTS

TIIVISTELMÄ
ABSTRACT
FOREWORD
TABLE OF CONTENTS
NOMENCLATURE
1 INTRODUCTION 7
2 JULIAFEM 8
2.1 FEMBeam 8
3 THEORY OF THE BEAM ELEMENT 10
4 NUMERICAL INTEGRATION 16
5 TEST MODEL 17
6 TESTING FEMBEAM.JL WITH TEST MODEL 18
7 CONCLUSIONS 19
8 SUMMARY 20
REFERENCES
APPENDICES
Appendix 1. Function for integration of a 2D beam elements stiffness matrix
Appendix 2. Function for integration of a 2D beam elements mass matrix
Appendix 3. Function for integration of a 2D beam elements force vector
Appendix 4. The test file for 2D beam element

NOMENCLATURE

A	Cross-section area
E	Young's modulus
$\boldsymbol{f}$	Nodal point forces vector
$\boldsymbol{f_q}$	Equivalent forces vector
I	Moment of inertia for z-axis
$\boldsymbol{k}$	Local stiffness matrix
l	Length of the element
$\boldsymbol{m}$	Local mass matrix
t	Time
$\boldsymbol{u}$	Displacement vector
$\ddot{\boldsymbol{u}}$	Acceleration vector
$\boldsymbol{X}_1$	Beam's left node's coordinates
$\boldsymbol{X}_2$	Beam's right node's coordinates
α	Rotation angle
ρ, *ro*	Density

DOF	Degree of freedom
FEM	Finite elemental method

1 INTRODUCTION

The aim for this work was to implement beam element into JuliaFEM – an open source finite element method solver written in the Julia code language. JuliaFEM is an ongoing project which develops software for reliable, scalable, distributed Finite Element Method. (JuliaFEM.jl Documentation 2018)

The beam element, more precisely Euler-Bernoulli beam element, was chosen because the theory and math behind it is not too complex. For this thesis the beam element is consider in only 2D cases but in future it will be implemented for also 3D cases. One goal for this thesis was introducing coding and JuliaFEM and that's why the 2D beam element was a reasonable choice. The Euler-Bernoulli beam theory, also known as engineering beam theory, is one of two major beam theories (Fish & Belytschko 2007, p. 249).

This thesis also briefly introduces Gaussian quadrature, which is widely used numerical integration method in finite element method.

The code was tested with an example frame model. Frame has three beam elements and two uniformly distributed loads which represent wind and snow loads on the structure.

2 JULIAFEM

Julia programming language is appropriate language for scientific and numerical calculations. Julia is a flexible dynamic language with performance of a traditional statically typed language. Julia may seem slow at first because of its compiler differs from many others but using it with the performance tips will make it faster. In the end Julia should be easy to write and almost as fast as the C language. (Julia documentation 2017)

JuliaFEM is an open-source finite element solver written in the Julia programming language. With JuliaFEM it's possible to process large finite element models across clusters of computers distributed from scale of single servers to thousands of machines. "The basic design principle is: Everything is nonlinear. All physics models are nonlinear from which linearizations are made as special cases". (Frondelius & Aho 2017)

JuliaFEM concept consists of several sub-packages and JuliaFEM itself is one package. Since JuliaFEM is coded as open-source it can be installed and contributed by anyone in Github. (Github 2018a)

2.1 FEMBeam

The FEMBeam.jl sub-package will be part of JuliaFEM concept. This package contains implementation of a beam element based on Euler-Bernoulli beam theory. The theory and implementation is introduced in this thesis. The repository for FEMBeam.jl package can be found at Github. (Github 2018b)

Fembeam.jl integrates local stiffness matrix, local mass matrix and local force vector for a beam element with given values by numerical integration, more precisely Gauss quadrature with five integration points. Four integration points would be enough for now but in the future, the cross-section values and distributed loads may be functions of x and five integration points is then needed.

For the function FEMBeam.jl given values are $\boldsymbol{X}_1$, $\boldsymbol{X}_2$, E, I, A, ro, q_1, q_2 and $\boldsymbol{f}$. $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ are vectors which include coordinates of the beam element's nodes. If the element is not vertical, $\boldsymbol{X}_1$ must always be the coordinates of the left node. E is the Young's modulus, I

is the moment of inertia, A is cross-section area, ro is density, q_1 is tangential distributed load, q_2 is normal distributed load and $\boldsymbol{f}$ is point forces vector. Distributed loads must be given in local coordinates since they are rotated with rotation matrix. Distributed loads must be uniformly distributed. Point forces vector must be given in global coordinates.

FEMBeam.jl is divided into three functions, one for stiffness matrix, one for mass matrix and one for equivalent forces vector. All three functions are included in this thesis and are shown in appendices 1-3.

3 THEORY OF THE BEAM ELEMENT

Euler-Bernoulli beam theory works well with case of a slender beam that is subjected only with lateral loads that cause small deflections. The theory has three assumptions. First assumption is that vertical displacement, the deflection, of the points on a cross-section are small and equal to the deflection of the beam axis. Second assumption is that there is only axial strain and torsional shear. Third assumption is that the cross-section does not deform in its plane and that it remains orthogonal to the beam axis after deformation. (Oñate 2013, p 1-2)

The 2D Euler-Bernoulli beam element has two nodes and four degrees of freedom. Both nodes have a DOF for deflection and its derivative rotation. The DOFs are numbered from left to right. Deflections are DOFs number one and three the rotations are DOFs number two and four. The DOFs are shown in the Figure 1.

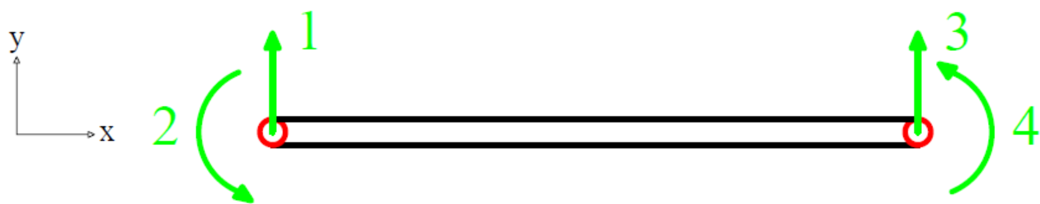

Figure 1. The 2D Euler-Bernoulli beam element's degrees of freedom.

The principle of virtual work leads to static equilibrium equation of the Euler-Bernoulli beam element which may be expressed as:

$$\int_0^L EI\boldsymbol{N}''^T\boldsymbol{N}''dx\,\boldsymbol{u} = \boldsymbol{f} + \int_0^L q_2\boldsymbol{N}^T\,dx, \tag{1}$$

where E is the Young's modulus of the beam, I is the area moment of inertia of the beam's cross section, $\boldsymbol{u}$ is the displacement vector, $\boldsymbol{f}$ is the force vector, q_2 is distributed load and $\boldsymbol{N}$ is vector of shape functions. (Oñate 2013, p. 9)

Equation (1) can be simplified as

$$\boldsymbol{ku} = \boldsymbol{f} + \boldsymbol{f_q}, \tag{2}$$

where $\boldsymbol{k}$ is the stiffness matrix and $\boldsymbol{f_q}$ is the equivalent forces vector.

Since the equation (1) includes second derivatives of the shape functions, the shape functions must be C^1-continuous. Hermite polynomials are one class of functions that provide the C^1-continuity (Fish & Belytschko 2007, p. 259).

The cubic Hermite splines expressed as shape functions:

$$\boldsymbol{N_1} = 1 - 3\frac{x^2}{l^2} + 2\frac{x^3}{l^3}, \tag{3}$$

$$\boldsymbol{N_2} = x\left(1 - 2\frac{x}{l} + \frac{x^2}{l^2}\right), \tag{4}$$

$$\boldsymbol{N_3} = \frac{x^2}{l^2}\left(3 - 2\frac{x}{l}\right) \text{ and} \tag{5}$$

$$\boldsymbol{N_4} = \frac{x^2}{l^2}\left(\frac{x}{l} - 1\right), \tag{6}$$

where $\boldsymbol{N_1}$ is the shape function of the DOF 1, $\boldsymbol{N_2}$ is the shape function of the DOF 2, $\boldsymbol{N_3}$ is the shape function of the DOF 3, $\boldsymbol{N_4}$ is the shape function of the DOF 4 and x is the x-coordinate.

From the integral on the left in equation (1) we get the stiffness matrix of the beam element. For example, if the moment of inertia and the Young's modulus are constant in the whole length of the beam the stiffness matrix can be written as:

$$k = \frac{EI}{l}\begin{bmatrix} 12/l^2 & 6/l & -12/l^2 & 6/l \\ 6/l & 4 & -6/l & 2 \\ -12/l^2 & -6/l & 12/l^2 & -6/l \\ 6/l & 2 & -6/l & 4 \end{bmatrix}. \tag{7}$$

The force vector has a component for every degree of freedom in the beam element. Nodal point forces and moments are contained in the force vector. The equivalent forces vector also has a component for every degree of freedom in the beam element. The distributed load is divided for every node by integration in the equivalent forces vector. If the distributed load is uniformly distributed the equivalent forces vector may be written as:

$$f_q = \begin{bmatrix} \frac{q_2 l}{2} \\ \frac{q_2 l^2}{12} \\ \frac{q_2 l}{2} \\ -\frac{q_2 l^2}{12} \end{bmatrix}, \tag{8}$$

where q_2 is uniformly distributed load per length unit. In dynamic problems the inertial forces become relevant and the displacements and forces become also functions of time. For the beam element we add the inertial forces to the equation (1). If damping is neglected the dynamic equilibrium equation for beam element may be written as:

$$\int_0^L \rho A \boldsymbol{N}^T \boldsymbol{N} dx\, \ddot{u} + \int_0^L EI \boldsymbol{N}''^T \boldsymbol{N}'' dx\, \boldsymbol{u} = \boldsymbol{f} + \int_0^L q \boldsymbol{N}^T\, dx, \tag{9}$$

where ρ is the density of the material, A is the cross-section area and $\ddot{\boldsymbol{u}}$ is the acceleration vector. (Hakala, 1986, p. 430-432)

Equation (9) may be simplified as:

$$\boldsymbol{m}\ddot{u} + \boldsymbol{k}\boldsymbol{u} = \boldsymbol{f} + \boldsymbol{f_q}\,, \tag{10}$$

where $\boldsymbol{m}$ is the mass matrix of the beam element. For example, if the cross-section values are constants in the whole length of the beam the mass matrix may be written after the integration as:

$$m = \frac{\rho A L}{420} \begin{bmatrix} 156 & 22l & 54 & -13l \\ 22l & 4l^2 & 13l & -3l^2 \\ 54 & 13l & 156 & -22l \\ -13l & -3l^2 & -22l & 4l^2 \end{bmatrix}. \tag{11}$$

This beam element with four degrees of freedom introduced above doesn't carry any axial forces and therefore is only suitable for pure deflection instances. Axial forces can be considered by adding linear 2D truss element's degrees of freedom to both nodes of the beam element as shown in Figure 2. However, these degrees of freedom are independent from the beam's degrees of freedom. This means that the stiffness and mass matrices get two more rows and columns for axial DOFs. For example, if the cross-section values are constants over the whole length of the beam the stiffness matrix can be written as:

$$k = \begin{bmatrix} \frac{EA}{L} & 0 & 0 & -\frac{EA}{L} & 0 & 0 \\ 0 & \frac{12EI}{l^3} & \frac{6EI}{l^2} & 0 & -\frac{12EI}{l^3} & \frac{6EI}{l^2} \\ 0 & \frac{6EI}{l^2} & \frac{4EI}{l} & 0 & -\frac{6EI}{l^2} & \frac{2EI}{l} \\ -\frac{EA}{L} & 0 & 0 & \frac{EA}{L} & 0 & 0 \\ 0 & -\frac{12EI}{l^3} & -\frac{6EI}{l^2} & 0 & \frac{12EI}{l^3} & -\frac{6EI}{l^2} \\ 0 & \frac{6EI}{l^2} & \frac{2EI}{l} & 0 & -\frac{6EI}{l^2} & \frac{4EI}{l} \end{bmatrix} \tag{12}$$

and the mass matrix can be written as:

$$m = \frac{\rho Al}{6} \begin{bmatrix} 2 & 0 & 0 & 1 & 0 & 0 \\ 0 & \frac{156}{70} & \frac{22l}{70} & 0 & \frac{54}{70} & -\frac{13l}{70} \\ 0 & \frac{22l}{70} & \frac{4l^2}{70} & 0 & \frac{13l}{70} & -\frac{3l^2}{70} \\ 1 & 0 & 0 & 2 & 0 & 0 \\ 0 & \frac{54}{70} & \frac{13l}{70} & 0 & \frac{156}{70} & -\frac{22l}{70} \\ 0 & -\frac{13l}{70} & -\frac{3l^2}{70} & 0 & -\frac{22l}{70} & \frac{4l^2}{70} \end{bmatrix}, \tag{13}$$

where A is the cross-section area of the beam.

The force vectors also get two more components for the axial DOFS. This combination of a beam and truss element can be used to analyze frame structures.

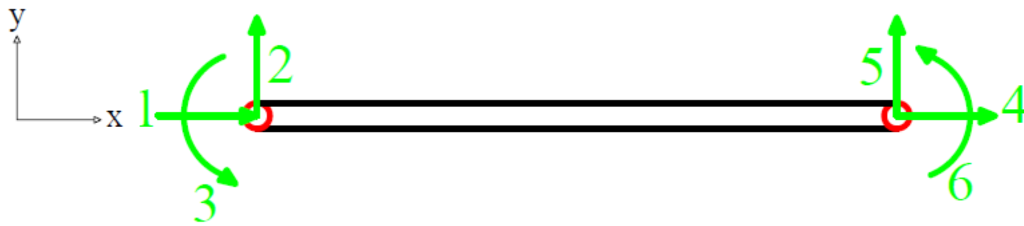

Figure 2. Beam element with six degrees of freedom.

If the beam element is rotated from the positive x-axis, the stiffness matrix, the mass matrix and the forces vector must be rotated using the rotation matrix $\boldsymbol{B}$ which may be written as:

$$\boldsymbol{B} = \begin{bmatrix} cos(\alpha) & sin(\alpha) & 0 & 0 & 0 & 0 \\ -sin(\alpha) & cos(\alpha) & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & cos(\alpha) & sin(\alpha) & 0 \\ 0 & 0 & 0 & -sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \tag{14}$$

where α is the rotation angle from the positive x-axis. The rotation angle is positive to counterclockwise direction as shown in Figure 3. FEMBeam.jl calculates the rotation angle with trigonometric function inverse tangent. There will be a division by zero inside the inverse tangent function if the beam is horizontal. In Julia code language if a floating-point value is divided by zero the output will be a special floating-point value infinite and inverse tangent gives 90° as output with infinite as input (Julia documentation 2017). The stiffness and mass matrices must be multiplied with transpose of the rotation matrix from the left side and with the rotation matrix from the right side. The equivalent forces vector must be multiplied from the left side with transpose of the rotation matrix.

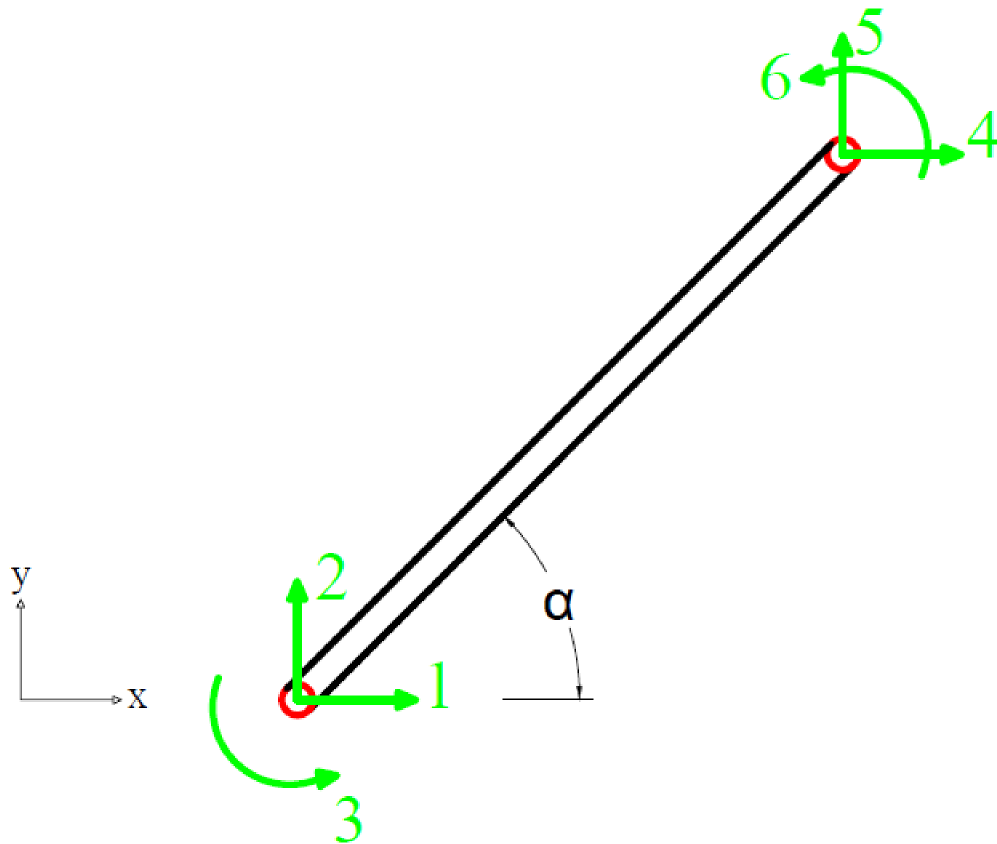

Figure 3. Rotated beam element with positive rotation angle.

In FEMBeam.jl the stiffness and mass matrices for truss DOFs and beam DOFs are integrated separately and assembled after integration to 6 DOF beam element's matrices as shown in appendices 1 and 2. Also the equivalent forces vector is integrated separately for truss DOFs and beam DOFs as shown in appendix 3.

4 NUMERICAL INTEGRATION

Different kinds of line, surface and volume integrals are calculated when forming finite element method matrices. Especially with elements that include high order functions the accurate integration is difficult if even possible. That's why numerical integration is necessary. In most cases it's also more efficient to use numerical integration than analytical integration in programming. (Hakala 1986, pages 304-306)

In numerical integration the integrand is evaluated at integration points and weighted sum of these values is used to approximate the integral. The integration points and weights are different in different numerical integration methods. (Hakala, 1986, pages 304-306)

The Gaussian quadrature is very efficient method for integrating polynomials. In finite element method the integrands usually involve polynomials and that's why it is reasonable to use Gaussian quadrature. The integration points and weights for every number of integration points are defined so that the integrand is as accurate as possible. By using Gaussian quadrature, it is possible to integrate exactly $2n-1$ -degree polynomials with n number of integration points. (Hakala 1986, p. 304-306)

In the beam element the shape functions are third degree Hermite polynomials. The stiffness matrix includes second derivatives of the shape functions. However, the mass matrix and the equivalent nodal forces vector include shape functions, not their derivatives. Stiffness matrix has the highest of second degree polynomials and equivalent forces vector has the highest of third degree polynomials so Gaussian quadrature with two integration points evaluates the integrals exactly. Mass matrix has the highest of 6th degree polynomial. With four integration points the beam element's stiffness and mass matrices and force vector is integrated exactly. The integration points and weights are presented as Gp and w in appendices 1-3.

5 TEST MODEL

The test model is a simple two-dimensional frame structure as shown in Figure 3. The structure is divided into three beam elements. Elements one and three have fixed supports to the ground. Structure has two uniformly distributed loads which represent wind and snow. Wind load is applied to element one and snow load is applied to element two.

In the test model all three beams have different material and cross-section values. Values for beam one are A_1 = 5160 mm^2, E_1 = 210 GPa, I_1 = 4.44 * 10^6 mm^4 and ρ_1 = 7800 kg/m^3. Values for beam two are A_2 = 7500 mm^2, E_2 = 250 GPa, I_2 = 6.25 *10^6 mm^4 and ρ_2 = 10000 kg/m^3. Values for beam three are A_3 = 2500 mm^2, E_3 = 160 GPa, I_3 = 521 *10^3 mm^4 and ρ_3 = 6000 kg/m^3. Uniformly distributed loads are q_{21} = -500 N and q_{22} = -750 N.

One feature of the code is to use rotation matrix to rotate matrices to the global coordinates. This is is tested by the test model with the two beam elements which are vertical. Element one has 90˚ rotation angle and element three has -90˚ rotation angle.

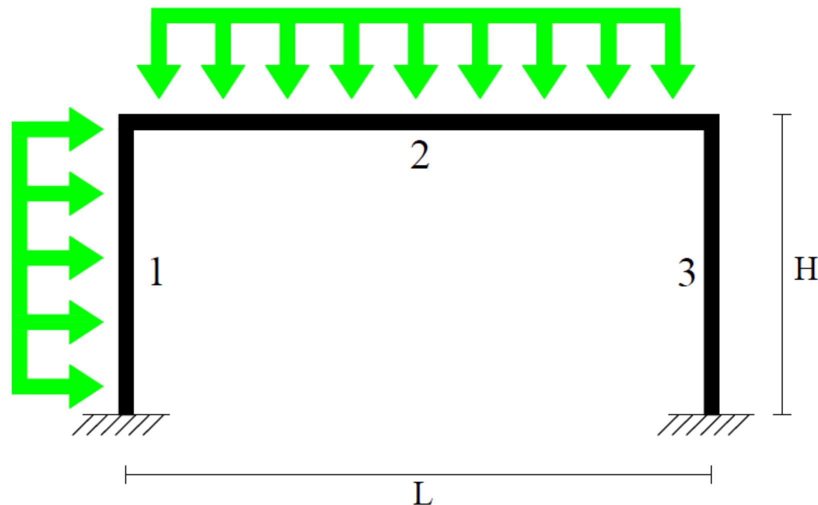

Figure 3. Test model.

6 TESTING FEMBEAM.JL WITH TEST MODEL

Like every package in JuliaFEM also FEMBeam.jl includes test files. Test files are made for packages to test them as widely as possible. The expected values in this case are results calculated by hand. In the test file a function called isapprox tests if two given values are approximately same with a 0,01% tolerance. In FEMBeam.jl's test files the expected stiffness matrix, expected mass matrix and expected force vector are called K_expected, M_expected and f_expected.

Local stiffness matrices, mass matrices and force vectors for each beam in the test model have one test in the test file. In all 12 tests values are calculated using FEMBeam.jl and the results are compared to results calculated by hand.

The results calculated by hand are presented in the test file in appendix 4.

7 CONCLUSIONS

Beam elements in the test model have different cross-section values, material values, rotation angles and lengths. In FEMBeam.jl all the matrices are integrated from shape functions with numerical integration. The FEMBeam.jl passes all the 12 tests in test file and it can be stated that FEMBeam.jl integrates stiffness matrix, mass matrix and force vector for uniformly distributed loads correctly. The rotation matrix works also correctly since two of the test model's beams are rotated from the x-axis and the FEMBeam.jl returns matrices in global coordinates correctly.

Next step for FEMBeam.jl would be to implement that cross-section and material values could variate in the length of the beam. Also the beam element should be expanded for three-dimensional problems.

8 SUMMARY

The goal for this work was to implement 2D beam element to JuliaFEM. The theory behind the element is Euler-Bernoulli beam theory which is one of two most common beam theories.

JuliaFEM as a concept and Julia code language were introduced briefly. Beam Theory and the formulation of stiffness matrix, mass matrix and the equivalent force vectors were presented closely. FEMBeam.jl uses Gaussian quadrature as the numerical integration method and it was also introduced.

The code was tested on a general frame structure with uniformly distributed loads. The structure's geometry was simple but the cross-section and material values were more complex to make the test convenient.

FEMBeam.jl works as planned. The code integrates desired matrices and vectors correctly and passes all tests. The two-dimensional beam element isn't very useful in practical use but after this implementation it should be reasonably easy to implement the 3D beam element.

REFERENCES

Fish, J. & Belytschko, T., 2007, A First Course in Finite Elements, Chichester, UK: John Wiley & Sons, Ltd, 344 p. ISBN 978-0-470-03580-1

Frondelius, T. & Aho, J., 2017. JuliaFEM – open source solver for both industrial and academia usage [online document]. Rakenteiden Mekaniikka, 50(3), p. 229-233. Available: https://doi.org/10.23998/rm.64224

Github, Inc, 2018a. JuliaFEM/JuliaFEM.jl [repository]. Available: https://github.com/JuliaFEM/JuliaFEM.jl [Referred 20.4.2018]

Github, Inc, 2018b. JuliaFEM/FEMBeam.jl [repository]. Available: https://github.com/JuliaFEM/FEMBeam.jl [Referred 19.5.2018]

Hakala, M., 1986, Lujuusopin elementtimenetelmä, Volume 3., Espoo, Finland: Otakustantamo, 490 p. ISBN 951-671-395-5

Julia documentation, 2018, introduction [online document]. Available: https://docs.julialang.org/en/stable/manual/introduction/ [Referred 1.4.2018]

Julia documentation, 2018, integers and Floating-point numbers [Online document]. Available: https://docs.julialang.org/en/release-0.4/manual/integers-and-floating-point-numbers/ [Referred 19.5.2018]

JuliaFEM documentation, 2018 [online document]. Available: http://www.juliafem.org/JuliaFEM.jl/latest/ [Referred 19.5.2018]

Oñate, E., 2013, Structural Analysis with the Finite Element Method, Linear statics: Volume 2: Beams, plates and shells, Heidelberg: Springer, 894 p. ISBN 978-1-4020-8742-4

APPENDICES

Appendix 1: Function for integration of a 2D beam element's stiffness matrix

Appendix 2: Function for integration of a 2D beam element's mass matrix

Appendix 3: Function for integration of a 2D beam element's forces vector

Appendix 4: The beam element's test file

Function for integration of a 2d beam elements stiffenss matrix

```
# This file is a part of JuliaFEM.
# License is MIT: see
https://github.com/JuliaFEM/FEMBeam.jl/blob/master/LICENSE

"""
Function integrates stiffness matrix for
6 DOF Euler-Bernoulli beam element.

    get_beam_stiffness_matrix_2d(X1,X2,E,I,A)

X1 = beams left node coordinates
X2 = beams right node coordinates
E = Young's modulus
I = Moment of inertia
A = Cross section area
"""
function get_beam_stiffness_matrix_2d(X1,X2,E,I,A)
    le=norm(X2-X1)                      # Lenght of element
    a=atan((X2[2]-X1[2])/(X2[1]-X1[1])) # Rotation angle of the
element
    nn=2                                # Number of nodes
    nd=3*1+3                            # Number of DOFs
    Gp=[-1/3*sqrt(5+2*sqrt(10/7)) -1/3*sqrt(5-2*sqrt(10/7)) 0
1/3*sqrt(5-2*sqrt(10/7)) 1/3*sqrt(5+2*sqrt(10/7))] # Five Gauss
integration points
    w=[(322-13*sqrt(70))/900 (322+13*sqrt(70))/900 128/225
(322+13*sqrt(70))/900 (322-13*sqrt(70))/900] # Five Gauss integration
weights
    # Rotation matrix
    B=[cos(a) sin(a) 0  0      0      0;
      -sin(a) cos(a) 0  0      0      0;
       0      0      1  0      0      0;
       0      0      0  cos(a) sin(a) 0;
       0      0      0 -sin(a) cos(a) 0;
       0      0      0  0      0      1]
    # Integration of the truss elements stiffness matrix
    detJ=2/le
    function tkint(wi::Float64, ::Float64)
        dN1=-1/2
        dN2=1/2
        dN=[dN1 dN2]
        return wi*dN'*dN
    end
    tk=zeros(2,2)
    for i = 1:size(Gp,2)
        tk +=tkint(w[i],Gp[i])
    end
    tk=detJ*A*E*tk
    # Integration of the 4 DOF beam elments stiffness matrix
    detJ=(2/le)^3
    function bkint(wi::Float64, xi::Float64)
        d2N1=(-1.0)*(1 - xi) + 0.5*(2 + xi)
        d2N2=(-1/2)*le*(1 - xi) + (1/4)*le*(1 + xi)
        d2N3=(-1.0)*(1 + xi) + 0.5*(2 - xi)
        d2N4=(1/4)*le*(-1 + xi) + (1/2)*le*(1 + xi)
```

```
        d2N=[d2N1 d2N2 d2N3 d2N4]
        return wi*d2N'*d2N
    end
    bk=zeros(4,4)
    for i = 1:size(Gp,2)
        bk +=bkint(w[i],Gp[i])
    end
    bk=E*I*detJ*bk
    # Assembly of 6 DOF truss-beam stiffness matrix k
    k=zeros(6,6)
    k[1,1]=tk[1,1]; k[1,4]=tk[1,2]; k[4,1]=tk[2,1]; k[4,4]=tk[2,2]
    k[2,2]=bk[1,1];k[2,3]=bk[1,2];k[2,5]=bk[1,3];k[2,6]=bk[1,4]
    k[3,2]=bk[2,1];k[3,3]=bk[2,2];k[3,5]=bk[2,3];k[3,6]=bk[2,4]
    k[5,2]=bk[3,1];k[5,3]=bk[3,2];k[5,5]=bk[3,3];k[5,6]=bk[3,4]
    k[6,2]=bk[4,1];k[6,3]=bk[4,2];k[6,5]=bk[4,3];k[6,6]=bk[4,4]
    # Rotation
    k=B'*k*B
    return k
end
```

Function for integrating a 2D beam element’s mass matrix

```
# This file is a part of JuliaFEM.
# License is MIT: see
https://github.com/JuliaFEM/FEMBeam.jl/blob/master/LICENSE

"""
Function integrates mass matrix for
6 DOF Euler-Bernoulli beam element in 2d.

    get_beam_mass_matrix_2d(X1,X2,A,ro)

X1 = beams left node coordinates
X2 = beams right node coordinates
A = Cross section area
ro = Density
"""
function get_beam_mass_matrix_2d(X1,X2,A,ro)
    le=norm(X2-X1)                        # Lenght of element
    a=atan((X2[2]-X1[2])/(X2[1]-X1[1])) # Rotation angle of the
element
    nn=2                              # Number of nodes
    nd=3*1+3                             # Number of DOFs
    Gp=[-1/3*sqrt(5+2*sqrt(10/7)) -1/3*sqrt(5-2*sqrt(10/7)) 0
1/3*sqrt(5-2*sqrt(10/7)) 1/3*sqrt(5+2*sqrt(10/7))] # Five Gauss
integration points
    w=[(322-13*sqrt(70))/900 (322+13*sqrt(70))/900 128/225
(322+13*sqrt(70))/900 (322-13*sqrt(70))/900] # Five Gauss integration
weights
    # Rotation matrix
    B=[cos(a) sin(a) 0  0      0       0;
      -sin(a) cos(a) 0  0      0       0;
       0      0      1  0      0       0;
       0      0      0  cos(a) sin(a)  0;
       0      0      0 -sin(a) cos(a)  0;
       0      0      0  0      0       1]
    # Integration of the truss mass matrix tm
    detJ=le/2
    function tmint(wi::Float64, xi::Float64)
        N1s=1+(-1/2)*(1+xi)
        N2s=(1/2)*(1 + xi)
        Ns=[N1s N2s]
        return wi*Ns'*Ns
    end
    tm=zeros(2,2)
    for i = 1:size(Gp,2)
        tm +=tmint(w[i],Gp[i])
    end
    tm=ro*A*tm*detJ
    # Integration of the beam mass matrix bm
    detJ=(le/2)
    function bmint(wi::Float64, xi::Float64)
        N1=1/4*(1-xi)^2*(2+xi)
        N2=le/8*(1-xi)^2*(xi+1)
        N3=1/4*(1+xi)^2*(2-xi)
        N4=le/8*(1+xi)^2*(xi-1)
        N=[N1 N2 N3 N4]
```

```
        return wi*N'*N
    end
    bm=zeros(4,4)
    for i = 1:size(Gp,2)
        bm +=bmint(w[i],Gp[i])
    end
    bm=ro*A*bm*detJ
    # Assembly of the 6 DOF truss-beam mass matrix m
    m=zeros(6,6)
    m[1,1]=tm[1,1];m[1,4]=tm[1,2];m[4,1]=tm[2,1];m[4,4]=tm[2,2]
    m[2,2]=bm[1,1];m[2,3]=bm[1,2];m[2,5]=bm[1,3];m[2,6]=bm[1,4]
    m[3,2]=bm[2,1];m[3,3]=bm[2,2];m[3,5]=bm[2,3];m[3,6]=bm[2,4]
    m[5,2]=bm[3,1];m[5,3]=bm[3,2];m[5,5]=bm[3,3];m[5,6]=bm[3,4]
    m[6,2]=bm[4,1];m[6,3]=bm[4,2];m[6,5]=bm[4,3];m[6,6]=bm[4,4]
    # Rotation
    m=B'*m*B
    return m
end
```

Function for integration of a 2D beam elements force vector

```
# This file is a part of JuliaFEM.
# License is MIT: see
https://github.com/JuliaFEM/FEMBeam.jl/blob/master/LICENSE

"""
Function integrates forces vector for
6 DOF Euler-Bernoulli beam element in 2D.

    get_beam_forces_vector_2d(X1,X2,qt,qn,f)

X1 = beams left node coordinates
X2 = beams right node coordinates
qt = Tangential uniformly distributed load
qn = Normal uniformly distributed load
f = Point forces vector in global coordinates
"""
function get_beam_forces_vector_2d(X1,X2,qt,qn,f)
    le=norm(X2-X1)                        # Lenght of element
    a=atan((X2[2]-X1[2])/(X2[1]-X1[1])) # Rotation angle of the
element
    nn=2                              # Number of nodes
    nd=3*1+3                             # Number of DOFs
    Gp=[-1/3*sqrt(5+2*sqrt(10/7)) -1/3*sqrt(5-2*sqrt(10/7)) 0
1/3*sqrt(5-2*sqrt(10/7)) 1/3*sqrt(5+2*sqrt(10/7))] # Five Gauss
integration points
    w=[(322-13*sqrt(70))/900 (322+13*sqrt(70))/900 128/225
(322+13*sqrt(70))/900 (322-13*sqrt(70))/900] # Five Gauss integration
weights
    # Rotation matrix
    B=[cos(a) sin(a) 0  0      0      0;
      -sin(a) cos(a) 0  0      0      0;
       0      0      1  0      0      0;
       0      0      0  cos(a) sin(a) 0;
       0      0      0 -sin(a) cos(a) 0;
       0      0      0  0      0      1]
    # Integration of the equivalent forces vector
    # For truss element tfq
    detJ=le/2
    function tfqint(wi::Float64, xi::Float64)
        N1s=1+(-1/2)*(1+xi)
        N2s=(1/2)*(1 + xi)
        Ns=[N1s N2s]
        return wi*Ns'
    end
    tfq=zeros(2,1)
    for i = 1:size(Gp,2)
        tfq +=tfqint(w[i],Gp[i])
    end
    tfq=qn*tfq*detJ
    # For 4 DOF beam element bfqe
    detJ=le/2
    function bfqint(wi::Float64, xi::Float64)
        N1=1/4*(1-xi)^2*(2+xi)
        N2=le/8*(1-xi)^2*(xi+1)
        N3=1/4*(1+xi)^2*(2-xi)
```

```
        N4=le/8*(1+xi)^2*(xi-1)
        N=[N1 N2 N3 N4]
        return wi*N'
    end
    bfq=zeros(4,1)
    for i = 1:size(Gp,2)
        bfq +=bfqint(w[i],Gp[i])
    end
    bfq=qt*bfq*detJ
    # Assembly of the 6 DOF beam element equivalent forces vector fqe
    fqe=zeros(6,1)
    fqe[1,1],fqe[4,1]=tfq[1,1],tfq[2,1]

fqe[2,1],fqe[3,1],fqe[5,1],fqe[6,1]=bfq[1,1],bfq[2,1],bfq[3,1],bfq[4,1
]
    # Rotation
    fqe=B'*fqe
    # Adding equivalent forces vector to point forces vector
    f +=fqe
    return f
end
```

The beam elements test file

```
# This file is a part of JuliaFEM.
# License is MIT: see
https://github.com/JuliaFEM/FEMBeam.jl/blob/master/LICENSE

using FEMBase
using FEMBeam

using Base.Test

@testset "Beam 1 Stiffness matrix" begin
X1=[0.0,0.0]; X2=[0.0,6.5]
E=210.0e9
I=50.8e-3*101.6e-3^3/12
A=50.8e-3*101.6e-3
k = FEMBeam.get_beam_stiffness_matrix_2d(X1,X2,E,I,A)
k_expected=
[40740.3     1.02079e-8  -1.32406e5    -40740.3    -1.02079e-8  -
1.32406e5;
 1.02079e-8  1.66749e8    8.10752e-12  -1.02079e-8 -1.66749e8
8.10752e-12;
-1.32406e5   8.10752e-12  573759.0      1.32406e5  -8.10752e-12
2.8688e5;
-40740.3    -1.02079e-8   1.32406e5     40740.3     1.02079e-8
1.32406e5;
-1.02079e-8 -1.66749e8   -8.10752e-12   1.02079e-8  1.66749e8   -
8.10752e-12;
-1.32406e5   8.10752e-12  2.8688e5      1.32406e5  -8.10752e-12
573759.0]
@test isapprox(k, k_expected, rtol=0.0001)
end

@testset "Beam 1 force vector" begin
X1=[0.0,0.0]; X2=[0.0,6.5]
qt=-500
qn=0
f=zeros(6,1)
f = FEMBeam.get_beam_forces_vector_2d(X1,X2,qt,qn,f)
f_expected=[1625.0, -9.95026e-14, -1760.42, 1625.0, -9.95026e-14,
1760.42]
@test isapprox(f, f_expected, rtol=0.0001)
end

@testset "Beam 1 mass matrix" begin
X1=[0.0,0.0]; X2=[0.0,6.5]
A=50.8e-3*101.6e-3
ro=7800
m = FEMBeam.get_beam_mass_matrix_2d(X1,X2,A,ro)
m_expected=
[97.1943 -6.10403e-16 -89.0948 33.6442 6.10403e-16 52.6469;
-6.10403e-16 87.2256 5.45548e-15 6.10403e-16 43.6128 -3.22369e-15;
-89.0948 5.45548e-15 105.294 -52.6469 3.22369e-15 -78.9703;
33.6442 6.10403e-16 -52.6469 97.1943 -6.10403e-16 89.0948;
6.10403e-16 43.6128 3.22369e-15 -6.10403e-16 87.2256 -5.45548e-15;
52.6469 -3.22369e-15 -78.9703 89.0948 -5.45548e-15 105.294]
@test isapprox(m, m_expected, rtol=0.0001)
```

```
end

@testset "Beam 2 Stiffness matrix" begin
X1=[0.0,6.5]; X2=[8.0,6.5]
E=250.0e9
I=75.0e-3*100.0e-3^3/12
A=75.0e-3*100.0e-3
k = FEMBeam.get_beam_stiffness_matrix_2d(X1,X2,E,I,A)
k_expected=
[2.34375e8 0.0 0.0 -2.34375e8 0.0 0.0;
0.0 36621.1 1.46484e5 0.0 -36621.1 1.46484e5;
0.0 1.46484e5 781250.0 0.0 -1.46484e5 390625.0;
-2.34375e8 0.0 0.0 2.34375e8 0.0 0.0;
0.0 -36621.1 -1.46484e5 0.0 36621.1 -1.46484e5;
0.0 1.46484e5 390625.0 0.0 -1.46484e5 781250.0]
@test isapprox(k, k_expected, rtol=0.0001)
end

@testset "Beam 2 force vector" begin
X1=[0.0,6.5]; X2=[8.0,6.5]
qt=-750
qn=0
f=zeros(6,1)
f = FEMBeam.get_beam_forces_vector_2d(X1,X2,qt,qn,f)
f_expected=[0.0, -3000.0, -4000.0, 0.0, -3000.0, 4000.0]
@test isapprox(f, f_expected, rtol=0.0001)
end

@testset "Beam 2 mass matrix" begin
X1=[0.0,6.5]; X2=[8.0,6.5]
A=75e-3*100e-3
ro=10000
m = FEMBeam.get_beam_mass_matrix_2d(X1,X2,A,ro)
m_expected=
[200.0 0.0 0.0 100.0 0.0 0.0;
0.0 222.857 251.429 0.0 77.1429 -148.571;
0.0 251.429 365.714 0.0 148.571 -274.286;
100.0 0.0 0.0 200.0 0.0 0.0;
0.0 77.1429 148.571 0.0 222.857 -251.429;
0.0 -148.571 -274.286 0.0 -251.429 365.714]
@test isapprox(m, m_expected, rtol=0.0001)
end

@testset "Beam 3 Stiffness matrix" begin
X1=[8.0,6.5]; X2=[8.0,0.0]
E=160.0e9
I=50.0e-3*50.0e-3^3/12
A=50.0e-3*50.0e-3
k = FEMBeam.get_beam_stiffness_matrix_2d(X1,X2,E,I,A)
k_expected=
[3641.33 -3.76792e-9 11834.3 -3641.33 3.76792e-9 11834.3;
-3.76792e-9 6.15385e7 7.24643e-13 3.76792e-9 -6.15385e7 7.24643e-13;
11834.3 7.24643e-13 51282.1 -11834.3 -7.24643e-13 25641.0;
-3641.33 3.76792e-9 -11834.3 3641.33 -3.76792e-9 -11834.3;
3.76792e-9 -6.15385e7 -7.24643e-13 -3.76792e-9 6.15385e7 -7.24643e-13;
11834.3 7.24643e-13 25641.0 -11834.3 -7.24643e-13 51282.1]
@test isapprox(k, k_expected, rtol=0.0001)
end

@testset "Beam 3 force vector" begin
```

```
X1=[8.0,6.5]; X2=[8.0,0.0]
qt=0
qn=0
f=zeros(6,1)
f = FEMBeam.get_beam_forces_vector_2d(X1,X2,qt,qn,f)
f_expected=zeros(6,1)
@test isapprox(f, f_expected, rtol=0.0001)
end

@testset "Beam 3 mass matrix" begin
X1=[8.0,6.5]; X2=[8.0,0.0]
A=50.0e-3*50.0e-3
ro=6000
m = FEMBeam.get_beam_mass_matrix_2d(X1,X2,A,ro)
m_expected=
[36.2143 2.27434e-16 33.1964 12.5357 -2.27434e-16 -19.6161;
2.27434e-16 32.5 2.03269e-15 -2.27434e-16 16.25 -1.20114e-15;
33.1964 2.03269e-15 39.2321 19.6161 1.20114e-15 -29.4241;
12.5357 -2.27434e-16 19.6161 36.2143 2.27434e-16 -33.1964;
-2.27434e-16 16.25 1.20114e-15 2.27434e-16 32.5 -2.03269e-15;
-19.6161 -1.20114e-15 -29.4241 -33.1964 -2.03269e-15 39.2321]
@test isapprox(m, m_expected, rtol=0.0001)
end
```