Final Report

Programutveckling för tekniska tillämpningar VSMN20

Authors: Kajsa Söderhjelm, Alexander Jörud

May 24, 2017

Contents

1 Introduction

The objective is to implement a finite element method solver based on CALFEM in python and to compare the solution from the solver with a developed finite element method solver produced in MATLAB using CALFEM. The solver is also to be implemented with a functioning gui (graphical working interface).

1.1 Problem formulation

The solver implemented is to be able to solve a plane stress finite element problem. The geometry that is to be investigated is depicted in figure 1 and can be related to a specimen exposed to uniaxial tensile stress. The parameters a, b, h and w can be altered by the end-user to solve the problem for different geometries. The line load q as well as the material properties modulus of elasticity E and poisson's ratio v may also be altered to modify the model.

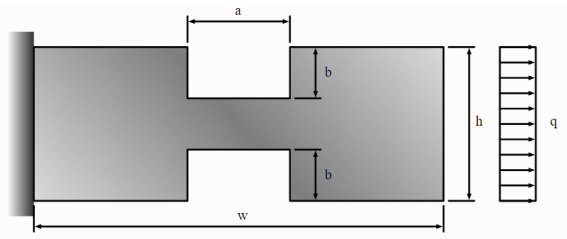


Figure 1: Represented is a illustration of the geometry, boundary condition and loads of the sample that the FEM software is built on.

In the developed software you should be able to perform a parameter study where the parameters a and b can be studied for different intervals and parameter steps. The software should be able to generate an illustration of the geometry, the mesh, the nodal displacements and the nodal values which in this case is the Von Mises element stress. Information about the user input data and the resulting output data is to be displayed in a console window for the user and the possibility to save the model to a file.

1.2 Limitations

There are several limitations with the software developed. It cannot handle inputs as zero quantities. Several geometric limitations exists in the software, e.g. the length b may not be altered to be greater than the height h divided by two, which is evident. Introducing errors as mentioned will result in issues when trying to generate the mesh and in turn when trying to solve the Finite Element problem.

2 Finite Element Method

Starting of with the strong form of equation of motion

$$\sigma_{ij,j} + b_i = \rho \ddot{u}_l \tag{1}$$

Rewriting equation (1) to

$$\int_V \left[(\sigma_{ij} v_i)_{,j} - \sigma_{ij} v_{i,j} \right] dV + \int_V \left[v_i b_i - \rho v_i \ddot{u}_l \right] dV \tag{2}$$

Using the divergence theorem and, v_i is an arbitrary vector not related to u_i, and defining $\epsilon^v_{ij} = \frac{1}{2}(v_{i,j} + v_{j,i})$ where ϵ^v_{ij} is related to the weight function v_i in the same manner as ϵ_{ij} is related to the displacement u_i symmetry of the stress tensor σ_{ij} provides

$$\int_V \rho v_i \ddot{u}_l dV + \int_V v_{i,j} \sigma ij dV = \int_S v_i t_i dS + \int_V v_i b_i dV \tag{3}$$

$$v_{i,j}\sigma_{ij} = \frac{1}{2}(v_{i,j}\sigma_{ij} + v_{j,i}\sigma_{ji}) = \frac{1}{2}(v_{i,j}\sigma_{ij} + v_{j,i}\sigma_{ij}) = \epsilon^v_{ij}\sigma_{ij} \tag{4}$$

Equation (3) and equation (4) provides the weak form of motion which is also known as the principle of virtual work

$$\int_V \rho v_i \ddot{u}_l dV + \int_V \epsilon_{ij}^v \sigma_{ij} dV = \int_S v_i t_i dS + \int_V v_i b_i dV \tag{5}$$

Where the quantities in equation (5) are defined bellow

$$\boldsymbol{\epsilon}^v = \begin{bmatrix} \epsilon_{11}^v \\ \epsilon_{22}^v \\ \epsilon_{33}^v \\ \epsilon_{12}^v \\ \epsilon_{13}^v \\ \epsilon_{23}^v \end{bmatrix} \qquad \boldsymbol{\sigma} = \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{12} \\ \sigma_{13} \\ \sigma_{23} \end{bmatrix} \qquad \ddot{\boldsymbol{u}} = \begin{bmatrix} \ddot{u}_1 \\ \ddot{u}_2 \\ \ddot{u}_3 \end{bmatrix} \qquad \boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \qquad \boldsymbol{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{6}$$

Where $\boldsymbol{v}$ the weight fuction, $\boldsymbol{t}$ is the traction vector and $\boldsymbol{b}$ the body force vector. With these notations equation (5) can be expressed as

$$\int_V \rho \boldsymbol{v}^T \ddot{\boldsymbol{u}} dV + \int_V (\boldsymbol{\epsilon}^v)^T \boldsymbol{\sigma} = \int_S \boldsymbol{v}^T \boldsymbol{t} dS + \int_V \boldsymbol{v}^T \boldsymbol{b} dV \tag{7}$$

To be able to express the displacement as an approximation through the entire body as a finite element method some notations needs to be established. $\boldsymbol{u} = \boldsymbol{u}(x_i, t)$ is the displacement vector, $\boldsymbol{N} = \boldsymbol{N}(x_i)$ us the global shape function, $\boldsymbol{a} = \boldsymbol{u}(t)$ is the nodal displacement, $\boldsymbol{v}$ is the weight function according to the Galerkin method. From this it follows

$$\ddot{\boldsymbol{u}} = \boldsymbol{N}\ddot{\boldsymbol{a}} \qquad \boldsymbol{B} = \boldsymbol{B}(x_i) = \frac{d\boldsymbol{N}}{dx_i} \qquad \boldsymbol{\epsilon} = \boldsymbol{B}\boldsymbol{a} \qquad \boldsymbol{v} = \boldsymbol{N}\boldsymbol{c} \tag{8}$$

The notations in (8) put into equation (7) and stating that $\boldsymbol{c}$ is arbitrary provides the notation for the finite element method. Assuming static conditions $\ddot{\boldsymbol{u}} = 0$ will result in that the equation of motion is reduced to the equilibrium conditions and becomes

$$\int_V \boldsymbol{B}^T \boldsymbol{\sigma} dV = \int_S \boldsymbol{N}^T \boldsymbol{t} dS + \int_V \boldsymbol{N}\boldsymbol{b} dV \tag{9}$$

For a linear elastic material the stress tensor can be approximated as $\boldsymbol{\sigma} = \boldsymbol{D}\boldsymbol{\epsilon} = \boldsymbol{D}\boldsymbol{B}\boldsymbol{a}$ where $\boldsymbol{D}$ is the constitutive matrix. For plane stress $\boldsymbol{D}$ becomes

$$\boldsymbol{D} = \frac{E}{1-v^2} \begin{bmatrix} 1 & v & 0 \\ v & 1 & 0 \\ 0 & 0 & 1-v \end{bmatrix} \tag{10}$$

Where E is the modulus of elasticity and v is the Poisson's ratio. This results in

$$(\int_V \boldsymbol{B}^T \boldsymbol{D}\boldsymbol{B} dV) * \boldsymbol{a} = \int_S \boldsymbol{N}^T \boldsymbol{t} dS + \int_V \boldsymbol{N}\boldsymbol{b} dV \tag{11}$$

Now the stiffness matrix $\boldsymbol{K}$, the load vector $\boldsymbol{f}$ are defined as

$$\boldsymbol{K} = \int_V \boldsymbol{B}^T \boldsymbol{D}\boldsymbol{B} dV \qquad \boldsymbol{f} = \int_S \boldsymbol{N}^T \boldsymbol{t} dS + \int_V \boldsymbol{N}\boldsymbol{b} dV \tag{12}$$

The element stiffness vector $\boldsymbol{K}$ is calculated with equation (11) using the existing CALFEM function `planqe` and assembled into the global stiffness matrix using the CALFEM function `assem`. Equation (11) and the definition in equation (12) provides

$$\boldsymbol{K} * \boldsymbol{a} = \boldsymbol{f} \tag{13}$$

The nodal displacements, $\boldsymbol{a}$ are to be solved from equation (13) where correct boundary conditions are applied.[1] The boundary conditions for the given problem will be applied in the far left side where the nodes are locked in both x- and y-direction. The line load and boundary conditions are applied with the CALFEM functions `applyforcetotal` and `applybc` respectivelyThe. The CALFEM function `solveq` is used to solve the equation system in equation (13).[2] Von Mises element stress for plane stress is calculated with

$$\sigma_v = \sqrt{\sigma_{11}^2 - \sigma_{11}\sigma_{22} + \sigma_{22}^2 + 3\tau_{12}} \tag{14}$$

The principal stress calculated with equation (15) and the principal directions are solved from that.

$$\sigma_{1,2} = \frac{\sigma_x + \sigma_y}{2} \pm \sqrt{\left(\frac{\sigma_x + \sigma_y}{2}\right)^2 - \sigma_x\sigma_y + \tau_{xy}^2} \tag{15}$$

3 Program Structure

In the following section of the report, the structure of the developed program is presented. The code is divided into `PlaneStress.py` and `Main.py`.

3.1 `Main.py`

This is the main python file for managing the main window (gui), which is observed by the user when running the program. It includes different classes and methods to be able to properly manage the different window applications such as buttons, a slide bar, etc. It is also in general connected to the other main python file, called the `PlaneStress`, with help of the different classes and methods. In more detail description of the different classes is presented below.

Note, not all methods are discussed for every class in the following section of the report, a choice of only discussing the more important methods has been made by the authors.

3.1.1 `SolverThread`

`SolverThread` is a class to manage calculations/computation of the solver in the background without "freezing the program" for the user. The `SolverThread` executes the solver from the `PlaneStress` python file parallel and in turn ensures that the loop of the application(`PyQt` application) of the main window does not "freeze" for the user. Since nothing will happen in the window until the solver method has finished unless the `SolverThread` method is implemented.

3.1.2 `MainWindow`

The `MainWindow` class manages the created window in `QtDesigner` and ensures with different methods that buttons, slide bar, drop down menu options and different text windows are properly connected and implemented. The class constructor which initiates the window produced in `QtDesigner`, connects buttons, slider bar, drop down menu options and different text windows.

- A `CalcDone` method is implemented to ensure certain buttons cannot be used until the user has executed the solver, i.e. cannot show the mesh until a simulation has been performed.
- The `onExecuteParamStudy` is connected to the `executeParamStudy` in the `PlaneStress.py` file. The method ensures that the user has "checked" one of the different parameters to study and successfully updates the values of either a or b to the controls (i.e `updateControls`) method.
- The `updateControls` method ensures the user input values in the gui are successfully updated.
- The `updateModels` obtains the values from the different parameters which may be altered by the user in the gui. They are transfered to the `inputData` class in `PlaneStress.py`.
- The `initModel` method initates a model which consists of default values of the parameters which may be altered in the gui. It also creates a necessary object for the `solver`.

3.2 `PlaneStress.py`

3.2.1 `InputData`

In the `InputData` class the geometry is created and contains the following different methods:

- The `Save` method ensures that certain parameters can be saved to a `json` file by the user.
- The `Load` method ensures that certain parameters can be loaded from a `json` file by the user.

3.2.2 `Solver`

In the `Solver` class the computations of the finite element method is performed and results such as Von Mises stress and nodal displacement is produced.

- The method `Execute`, loads certain parameters inserted by the user in the window application necessary to perform a FEA (Finite Element Analysis) and computes nodal displacement and Von Mises stress. Settings such as element type, degrees of freedom and boundary conditions are implemented here. Additional results exported is the location of the element which possess the maximum Von Mises Stress ,the size of the maximum Von Mises Stress and principle stresses.
- The method `executeParamStudy` performs a parameter study by altering either the a parameter or b parameter defined in figure 1.
- The `exportVtk` method makes sure a `.vtk` file is exported and in turn can be imported to a software called ParaView, where further visualization of the results can be performed.

3.2.3 `OutputData`

`OutputData` class contains a constructor which stores the produced results from the execute method in the `Solver` class.

3.2.4 `Report`

The `Report` class is implemented to produce a report of the initial parameters and results from the solver method, i.e from the `InputData` and `OutputData` methods in the `Solver` class. The report is connected to the `Main` python file which enables the user to observe the report in a "console" window. An additional method not necessary for completing the main task is implemented, called `writeToFile`. The `writeToFile` method is implemented in case the user desires to save the produced report in the "console" window to a text file (`.txt`).

3.2.5 `Visualisation`

The `Visualisation` class enables the user to visualize different results based on user input from the window application in the `Main.py` file. It is connected to different buttons and the following visualization methods are implemented:

- The `showGeometry` method "calls" on the geometry stored in the `OutputData` class from the which originates from the `InputData` class.
- The `showMesh` method visualizes the geometry for the user. The mesh is stored in the `OutputData` class, which originiates from the `Solver` class, where the mesh is generated.
- The `showNodalDisplacement` ensures the user has the possibility to display the nodal displacement. The nodal displacement is a result output stored in the `OutputData` class from solving the FEA problem.
- The `showElementValues` method ensure the user has the ability to display the Von Mises stress, also originate from solving the FEA problem.

4 Example

4.1 Input parameters

The load and parameters on the sample depicted in figure 2 are to be analyzed using finite element analysis. All the parameters used can be obtained from table 1.

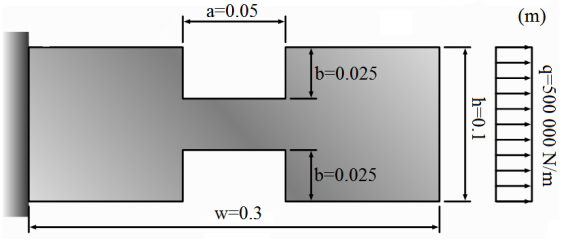


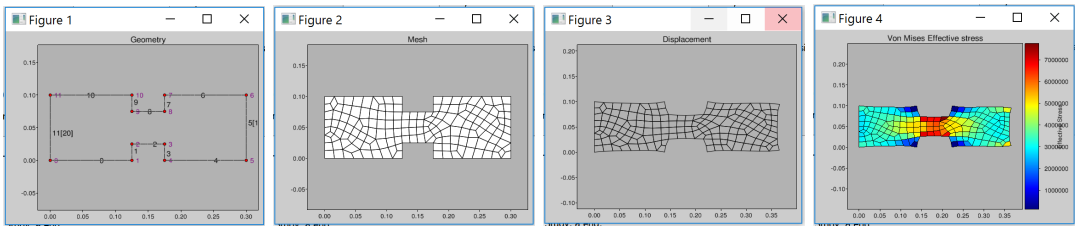
Figure 2: Example where the geometry parameters and line load are depicted.

w	0.3 m	E	20.8 GPa
h	0.1 m	v	0.33
a	0.05 m	q	500 000 N/m
b	0.025 m		

Table 1: My caption

4.2 Result

Graphical result of the analyses is presented in figure 3 and the printed result is presented in figure 4 where the magnification factor, nodal displacement, maximum and minimum von Mises element stress and maximum and minimum displacement can be retrieved.



Geometry Mesh Nodal displacement Von Mises stress

Figure 3: Graphical presentation of the geometry, mesh, nodal displacement and von Mises element stress distribution

.

```
---------- Model output ---------------------------------

The size of the magnification factor used for the visual results of nodal displacements:

1000

Nodal displacements, elementwise:

[[ 1.15454235e-05 -1.24973082e-06  1.13618645e-05 ..., -2.25181570e-06
   8.74752554e-06 -1.00043424e-06]
 [ 5.05818263e-05  9.79918740e-07  5.04782152e-05 ...,  1.72094399e-06
   5.33688760e-05  5.85674512e-07]
 [ 4.03746599e-05  1.11724500e-06  4.36554760e-05 ...,  1.32712949e-06
   3.99378839e-05  1.03735146e-06]
 ...,
 [ 4.83930230e-05 -1.38946556e-07  5.09363109e-05 ..., -8.09900261e-07
   4.99666258e-05 -1.23055706e-06]
 [ 1.10362054e-05  9.83960848e-07  1.16403833e-05 ..., -1.73115622e-07
   9.10159946e-06  4.36194381e-07]
 [ 5.35742832e-05 -4.86440433e-07  5.19354001e-05 ...,  4.06086299e-09
   5.25444595e-05 -4.84673939e-09]]

Max Von Mises stress (Pa):

7743607.05545

Min Von Mises stress (Pa):

117865.024531

Max Nodal displacement (m):

6.58785540009e-05

Min Nodal displacement (m):

0.0
```

Figure 4: The printed result

5 Comparing Matlab and Python solutions

5.1 Python code

```
# -*- coding: utf-8 -*-
"""

@author: Kajsa and Alex
"""

import numpy as np
import calfem.core as cfc
import json
```

```
class InputData(object):
    """Class to define the inputdata for the model."""

    def __init__(self):

        self.version = 1

        self.t = 0.01        #Thickness [m]
        self.ptype = 1       #Plan stress

        self.ep = [self.ptype, self.t]

        # --- Elementegenskaper

        self.E = 20.8*10**9 #Young's modulus
        self.v = 0.3        #Poisson's tal


        self.hooke = cfc.hooke(self.ptype, self.E, self.v) #Constitutive matrix

        # --- Skapa indata

        self.coord = np.array([
            [0.0, 0.0],
            [0.0, 0.1],
            [0.1, 0.0],
            [0.1, 0.1],
            [0.2, 0.0],
            [0.2, 0.1]
        ])

        self.dof = np.array([
            [1, 2],
            [3, 4],
            [5, 6],
            [7, 8],
            [9, 10],
            [11, 12]
        ])


        # --- Elementtopolgi

        self.edof = np.array([
            [1 , 2 , 7 , 8  , 3  , 4],
            [1 , 2 , 5 , 6  , 7  , 8],
            [5 , 6 , 11 , 12 , 7 , 8],
            [5 , 6 , 9 , 10 , 11 , 12]
        ])


        # --- Laster

        self.loads = np.array([
            [12, -10*10**3]
        ])

        # --- Randvillkor
```

```
        self.bcs = np.array([
            [1, 0],
            [2, 0],
            [3, 0],
            [4, 0]
        ])

    def save(self, filename):
        """Save inputdata to file."""

        inputData = {}
        inputData["version"] = self.version
        inputData["t"] = self.t
        inputData["ptype"]= self.ptype
        inputData["Young's modulus, E"] = self.E
        inputData["Poisson's ratio, v"] = self.v


        inputData["coord"] = self.coord.tolist()
        inputData["dof"] = self.dof.tolist()
        inputData["edof"] = self.edof.tolist()
        inputData["loads"] = self.loads.tolist()
        inputData["bcs"] = self.bcs.tolist()


        ofile = open(filename, "w")
        json.dump(inputData, ofile, sort_keys = True, indent = 4)
        ofile.close()


    def load(self, filename):
         """Read inputdata from file."""

        ifile = open(filename, "r")
        inputData = json.load(ifile)
        ifile.close()

        self.version = inputData["version"]
        self.t = inputData["t"]
        self.E = inputData["Young's modulus, E"]
        self.v = inputData["Poisson's ratio, v"]
        self.ptype = inputData["ptype"]
        self.ep = [self.ptype, self.t]
        self.hooke = cfc.hooke(self.ptype, self.E, self.v) #Constitutive matrix

        self.coord = np.asarray(inputData["coord"])
        self.dof = np.asarray(inputData["dof"])
        self.edof = np.asarray(inputData["edof"])
        self.loads = np.asarray(inputData["loads"])
        self.bcs = np.asarray(inputData["bcs"])

class Solver(object):
    """Class to manage the solution of the model"""
    def __init__(self, inputData, outputData):
        self.inputData = inputData
        self.outputData = outputData


    def execute(self):

        # Transfer to model parameters
```

```
            edof = self.inputData.edof
            coord = self.inputData.coord
            dof = self.inputData.dof
            ep = self.inputData.ep
            loads = self.inputData.loads
            bcs = self.inputData.bcs
            D = self.inputData.hooke



            ndof = edof.max()
            K = np.zeros([ndof,ndof])          #Stiffness matrix
            f = np.zeros([ndof,1])             #External force vector
            f[loads[0][0]-1] = loads[0][1]     #Add prescribed load
            es = np.zeros([edof.shape[0],3])   #Stress vector
            bc = bcs[:,0]

            ex, ey = cfc.coordxtr(edof,coord,dof)


            for elx, ely, Edof in zip(ex, ey, edof):
                Ke = cfc.plante(elx,ely,ep,D)
                cfc.assem(Edof,K,Ke)


            [a,r] = cfc.solveq(K,f,bc)
            ed = cfc.extractEldisp(edof,a)


            i = 0
            for elx, ely, Ed in zip(ex, ey, ed):
                [es[i], _ ] = cfc.plants(elx, ely, ep, D, Ed)

                i+=1

            # Outputdata
            self.outputData.a = a
            self.outputData.r = r
            self.outputData.ed = ed
            self.outputData.es = es

class OutputData(object):
    """Class to store the results from the simulation"""
    def __init__(self):
        self.a = None
        self.r = None
        self.ed = None
        self.es = None




class Report(object):
    """Class for presentation of inputdata and outpdata in a report "structure" """
    def __init__(self, inputData, outputData):
        self.inputData = inputData
        self.outputData = outputData
        self.report = ""
```

```
def clear(self):
    self.report = ""

def addText(self, text=""):
    self.report+=str(text)+"\n"

def save(self, filename):
    """Save outputdata to file."""

    outputData = {}
    outputData["Displacements (m):"] = self.outputData.ed.tolist()
    outputData["Stress (Pa):"] = self.outputData.es.tolist()


    ofile = open(filename, "w")
    json.dump(outputData, ofile, sort_keys = True, indent = 4)
    ofile.close()


def load(self, filename):
    """Read data from file."""

    ifile = open(filename, "r")
    inputData = json.load(ifile)
    ifile.close()

    self.version = inputData["version"]
    self.t = inputData["t"]


    self.coord = np.asarray(inputData["coord"])


def __str__(self):
    self.clear()
    self.addText()
    self.addText("-------------- Model input ---------------------------------")
    self.addText()

    self.addText("Thickness (m):")
    self.addText()
    self.addText(self.inputData.t)
    self.addText()

    self.addText("Young's modulus (Pa):")
    self.addText()
    self.addText(self.inputData.E)
    self.addText()

    ptype_temp = self.inputData.ptype
    self.addText("Type of material matrix:")
    self.addText()

    if ptype_temp == 1:

        self.addText("Plane stress")
        self.addText()
    elif ptype_temp == 2:

        self.addText("Plane strain")
```

```
            self.addText()
        elif ptype_temp == 3:

            self.addText("Axisymmetry")
            self.addText()

        else:

            self.addText("Three dimensional")
            self.addText()


        self.addText("Constitutive matrix:")
        self.addText()
        self.addText(self.inputData.hooke)
        self.addText()

        self.addText("Coordinates:")
        self.addText()
        self.addText(self.inputData.coord)
        self.addText()

        self.addText("Topology:")
        self.addText()
        self.addText(self.inputData.edof)
        self.addText()



        self.addText()
        self.addText("-------------- Results ----------------------------------")
        self.addText()

        self.addText("Displacements (m):")
        self.addText()
        self.addText(self.outputData.ed )
        self.addText()

        self.addText("Stress (Pa):")
        self.addText()
        self.addText(self.outputData.es)
        self.addText()

        return self.report

# -*- coding: utf-8 -*-
"""

@author: Kajsa and Alex
"""

import StressCal as SC


if __name__ == "__main__":

    inputData = SC.InputData()
    outputData = SC.OutputData()

    solver = SC.Solver(inputData,outputData)
    solver.execute()
```

```
report = SC.Report(inputData,outputData)
print(report)

#Spara inputdata
file_inputData = "inputData_file"
inputData.save(file_inputData)

#Spara (viktig) outputdata
file_outputData = "outputData_file"
report.save(file_outputData)
```

5.2 Python result


```
{
    "Displacements (m):": [
        [
            0.0,
            0.0,
            9.912071417847012e-05,
            -0.00021679795949008617,
            0.0,
            0.0
        ],
        [
            0.0,
            0.0,
            -0.00010539736105422796,
            -0.0002377201157426126,
            9.912071417847012e-05,
            -0.00021679795949008617
        ],
        [
            -0.00010539736105422796,
            -0.0002377201157426126,
            0.00012136238773775094,
            -0.0005898351191941023,
            9.912071417847012e-05,
            -0.00021679795949008617
        ],
        [
            -0.00010539736105422796,
            -0.0002377201157426126,
            -0.00012736345306196423,
            -0.0005678979496286879,
            0.00012136238773775094,
            -0.0005898351191941023
        ]
    ],
    "Stress (Pa):": [
        [
            22656163.24079317,
            6796848.97223795,
            -17343836.759206895
        ],
        [
            -22656163.24079315,
            -2445040.4717124514,
            -2656163.240793159
        ],
        [
            6518473.24229457,
            6307350.473213863,
```

```
            -13481526.757705448
        ],
        [
            -6518473.2422945555,
            -6518473.242294579,
            -6518473.242294563
        ]
    ]
}
```

5.3 Matlab code

```
clc
clear all
close all

%Material parameters
E = 20.8e9;
v = 0.3;
ptype = 1;
t = 0.01;

%Compute constitutive matrix, hooke.
D = hooke(ptype,E,v);

edof = [1, 1 , 2 , 7 , 8  , 3  , 4
        2, 1 , 2 , 5 , 6  , 7  , 8
        3, 5 , 6 , 11 , 12 , 7 , 8
        4, 5 , 6 , 9 , 10 , 11 , 12];

coord = [0.0, 0.0
         0.0, 0.1
         0.1, 0.0
         0.1, 0.1
         0.2, 0.0
         0.2, 0.1];


  dof =   [1, 2
           3, 4
           5, 6
           7, 8
           9, 10
           11, 12];

nen = 3;


[ex,ey]=coordxtr(edof,coord,dof,nen);

ndof = max(max(edof));
nelm = max(edof(:,1));
K = zeros(ndof);
f = zeros(ndof,1); %External force vector
f(12) = -10000;
es = zeros(nelm,nen);
ep = [ptype t];
eq = [0;0];
bc = [1, 0
      2, 0
      3, 0
      4, 0];
```

```
for i=1:nelm

        [ Ke ] = plante( ex(i,:),ey(i,:),ep,D,eq);


         indx = edof(i,2:end);
         K(indx,indx) = K(indx,indx)+Ke;
end

a = solveq(K,f,bc);
ed = extract(edof,a);



for z=1:nelm

        [es(z,:),~] = plants(ex(z,:),ey(z,:),ep,D,ed(z,:));

end

figure
eldisp2(ex,ey,ed,[1 6 0],5);
grid on

hold on

eldraw2(ex,ey,[1 2 0])

disp('Element stress components:')
disp(es)
```

5.4 Matlab results

```
Element stress components:
   1.0e+07 *

    2.2656    0.6797   -1.7344
   -2.2656   -0.2445   -0.2656
    0.6518    0.6307   -1.3482
   -0.6518   -0.6518   -0.6518
```

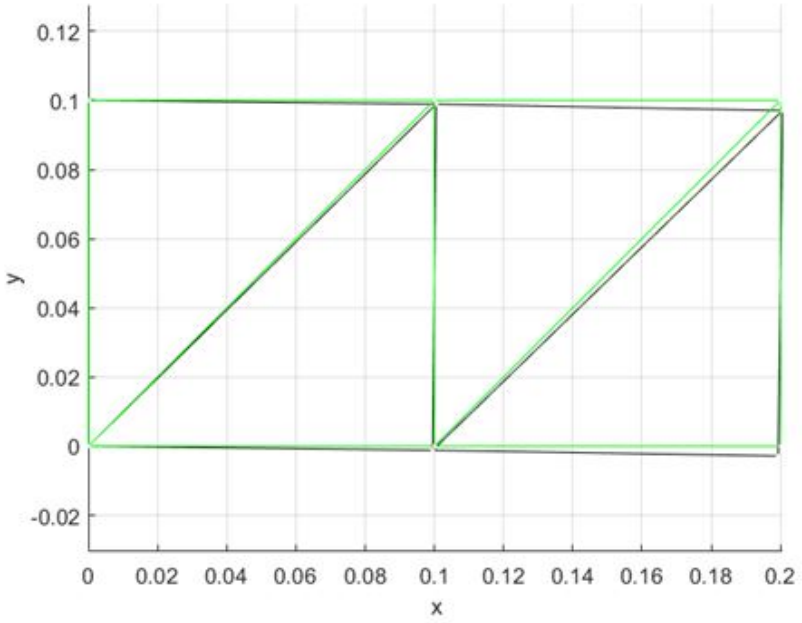


Figure 5: Matlab results, where the displaced elements are represented by the red color and the non-displaced elements are represented by the green color.

5.5 Plausibility assessment

The results seems reliable and are exact when compared to each other. The solution of stresses between python and Matlab are exact to each other and a further evaluation of the displacements, depicted in figure 5, ensures an accurate solution of the Finite Element solver. In figure 5, the structure seems to behave in a satisfactory manner according to how the load onto the structure is defined (top right node in negative y-direction in figure 5).

References

[1] Niels Saabye Ottosen and Matti Ristinmaa. *The Mechanics of Constitutive Modeling.* Elsevier Science Ltd, 2005.

[2] J Lindemann A Olsson K-G Olsson K Persson H Petersson M Ristinmaa G Sandberg P-A Wernberg P-E Austrell, O Dahlblom. *CALFEM, a finite element toolbox*, volume 3.4. 2004.

6 Manual

The geometry, boundary conditions and load case which the software will solve using the finite element method is depicted below in figure 1. The parameters a, b, w and h will alter the geometry, observe that all the parameters must have a value above zero. The magnitude of the line load q can also be changed.

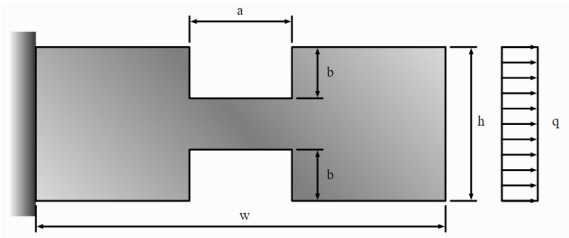


Figure 1: Geometry, boundary conditions and line load.

The geometric parameters including the thickness, modulus of elasticity, poisson's ratio and the magnitude of the line load can be altered according to the user by changing these in the boxes seen in figure 2 When **clicking** on "File", in the upper right corner, a drop down menu becomes available where it is possible to create a new model, open a saved model, save the current model and exit the software.

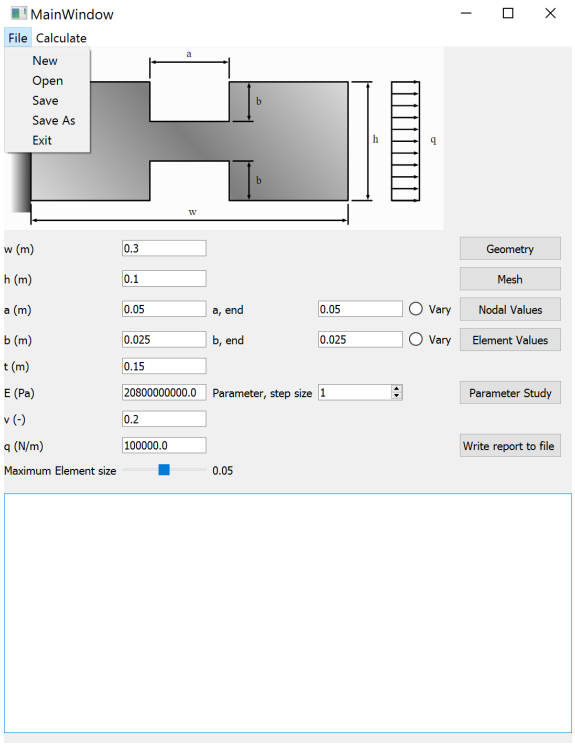


Figure 2: The main window where the drop down menu "File" is displayed.

If you select "New" you will be given a choice if you want to create a new model or not, observed in figure 3. If you select yes, a new model will be created. If the model is not saved before this stage, you will have to choose if you want to save the model or not.

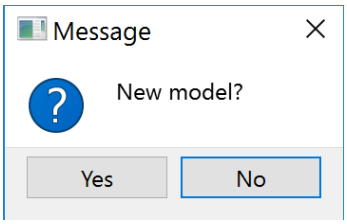


Figure 3: Creating a new model.

If you select "Open", you will be able to select an existing file from you computer, depicted in figure 4. The file should be a .json-, .jpg- or .bmp-file.

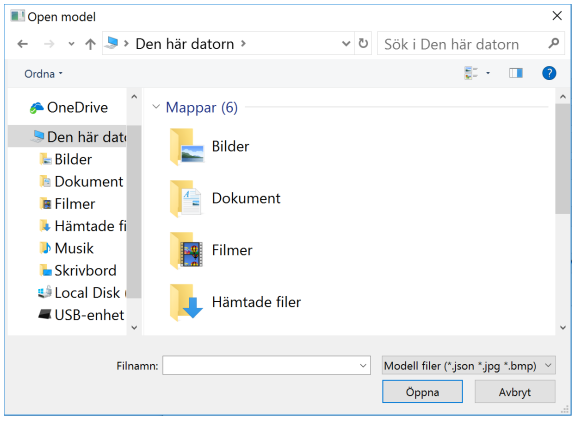


Figure 4: Open a existing file from your computer.

If you would like to save a model that you are working with to your computer select "Save" in the drop down menu from "File" in the upper left corner. If you are currently working with an existing file, the file will be updated. If there is no existing file you will be able to save a new file instead, see figure 5. If you want to save a new file, or a copy of the file, select "Save As".

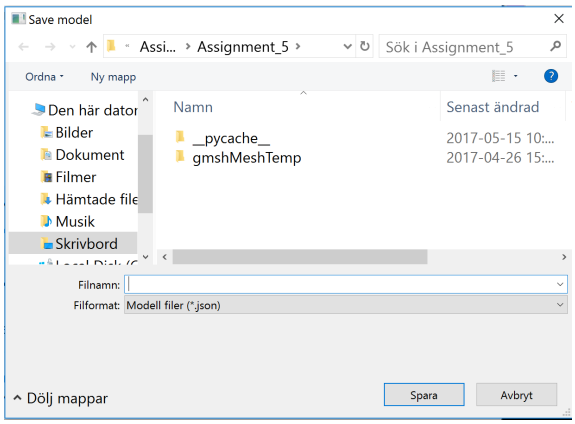


Figure 5: Save a file to your computer.

If you want to exit and close the program, select "Exit" in the drop down menu from "File" in the upper left corner.

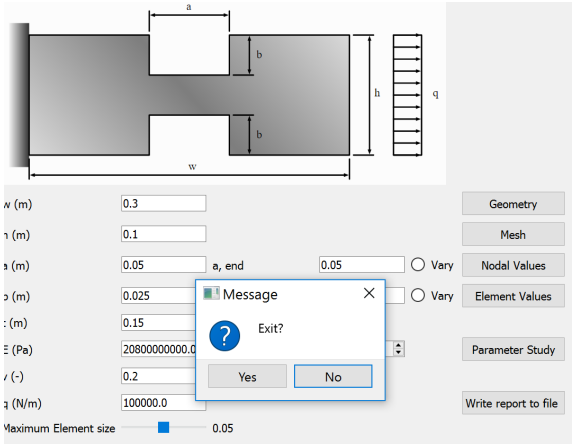


Figure 6: Exit and close the program.

In order to execute the calculations and solve the FE-problem select "Calculate" and then "Execute". You will be given a choice whether you want to run the simulation or do nothing. This can

be observed in figure 7.

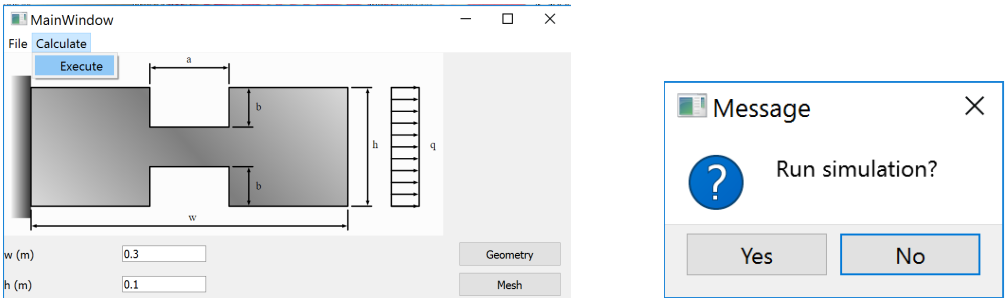


Figure 7: Executing the simulation to solve the problem.

A visual result of the problem and solution can be retrieved after the simulation has been executed. The visualization of the geometry, mesh, nodal displacement and Von Mises Element stress can be retrieved by selecting respectively button, seen in figure 8.

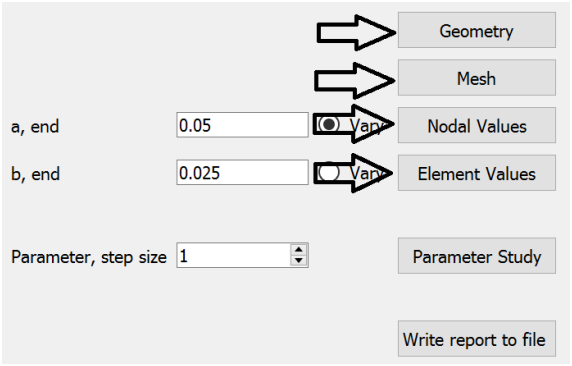
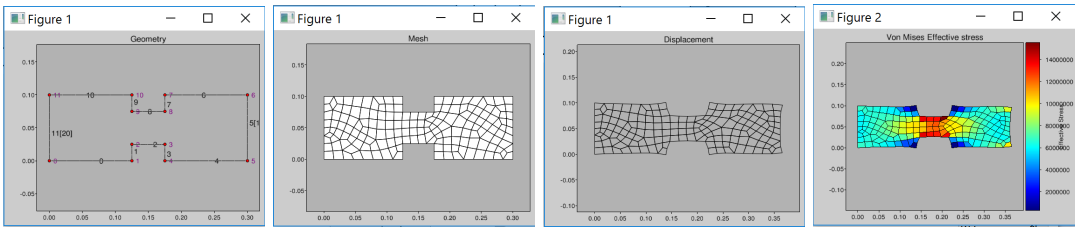


Figure 8: The buttons that can be selected in order to show the geometry, the mesh, the nodal displacement and Von Mises Element stress.



Geometry Mesh Nodal displacement Von Mises stress

Figure 9: Illustration of the types of visualization that can be retrieved.

.

In order to save the report to a text-file (`.txt`), press the "Write report to file"-button and you will need to enter a file name and press "Save". This is seen in figure 10

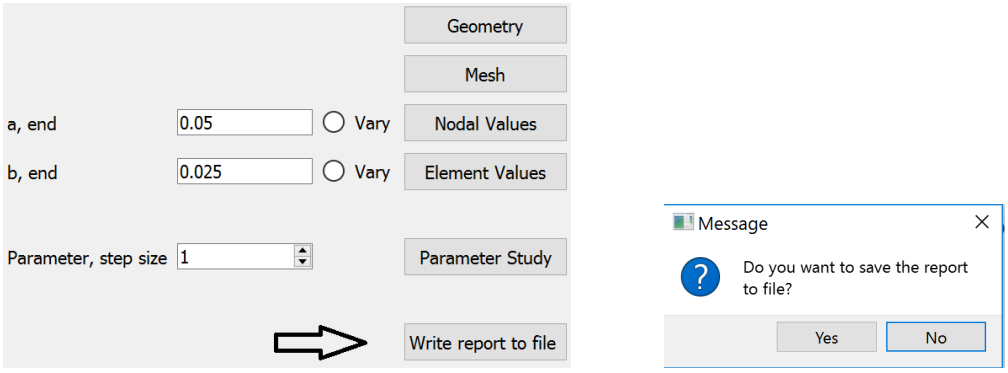


Figure 10: Saving the report to a text-file.

.

A parameter study of either the parameter a or the parameter b can be done. Start by choosing which parameter should vary and enter a starting value and ending value of this parameter, depicted in figure 11.

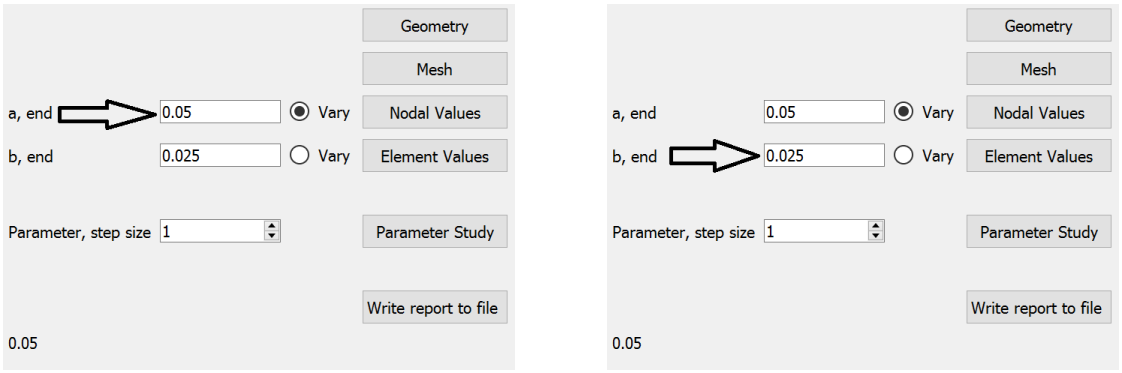


Figure 11: Chose a starting value and ending value of the parameter you wish to do a study on.

After a starting point and a ending point for either the parameter a or parameter b and the "Vary"-button is activated a parameter step size should be chosen. The step size can be 1-20 steps and decides the interval steps when running the parameter study. To execute the parameter study, press on the parameter study button, see figure 12.

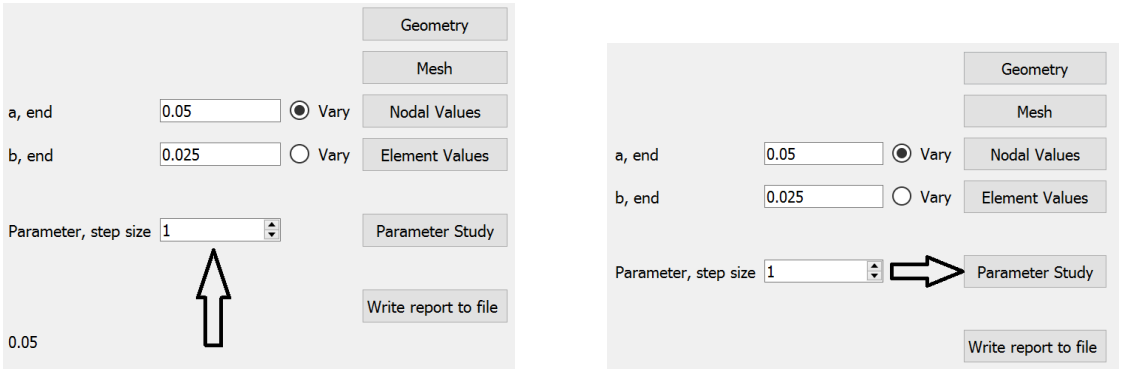


Figure 12: Enter the interval step size and execute the parameter study.

The files extracted from the parameter study can be imported to the software "ParaView" for further investigation of the results.

7 Python code

7.1 Main.py

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr  3 10:41:44 2017

@author: kajsa
"""

import sys

from PyQt5.QtWidgets import QApplication, QMainWindow, QFileDialog,QMessageBox
from PyQt5.uic import loadUi

import PlaneStress as ps

from PyQt5 import QtCore

class SolverThread(QtCore.QThread):
    """To manage the solver in the background"""

    def __init__(self, solver, paramStudy=False):
        """Class Constructor"""
        QtCore.QThread.__init__(self)
        self.solver = solver
        self.paramStudy = paramStudy

    def __del__(self):
        self.wait()

    def run(self):

        if self.paramStudy == True:
            self.solver.executeParamStudy()
        else:
            self.solver.execute()


class MainWindow(QMainWindow):
    """MainWindow-klass which manages the main window"""

    def __init__(self):
        self.filename = None
        self.filenameReport = None

        super(QMainWindow, self).__init__()

        # Store a reference to the applicationinstance in the class
        self.app = app

         # Read an interface from file
        self.ui = loadUi('mainWindow.ui', self)


        # Connect controls to an eventmethod
        # To connect events in the menu: Triggered
        # To connect buttons: Clicked

        """Menu"""
```

```
        self.ui.actionNew.triggered.connect(self.onActionNew)

        self.ui.actionOpen.triggered.connect(self.onActionOpen)

        self.ui.actionSave.triggered.connect(self.onActionSave)

        self.ui.actionSaveAs.triggered.connect(self.onActionSaveAs)

        self.ui.actionExit.triggered.connect(self.onActionExit)

        self.ui.actionExecute.triggered.connect(self.onActionExecute)


        """Buttons"""
        self.ui.showGeometryButton.clicked.connect(self.onShowGeometry)

        self.ui.showMeshButton.clicked.connect(self.onShowMesh)

        self.ui.showNodalValuesButton.clicked.connect(self.onShowNodalValues)

        self.ui.showElementValuesButton.clicked.connect(self.onShowElementValues)

        self.ui.WriteOutPutButton.clicked.connect(self.onWriteReport)

        self.ui.showParamButton.clicked.connect(self.onExecuteParamStudy)


        """Slider"""
        self.ui.ElemSlider.setRange(0,100)
        self.ui.ElemSlider.setSingleStep(1)
        self.ui.ElemSlider.valueChanged.connect(self.onValueChanged)

        # Ensure to display the window
        self.ui.show()
        self.ui.raise_()



    def onSolverFinished(self):
        """Called when the calculation thread ends"""

        # --- Activate inteface

        self.ui.setEnabled(True)

        # --- Generate result report
        self.report = ps.Report(self.inputData,self.outputData)

        self.ui.reportEdit.setPlainText(str(self.report))


    def onActionNew(self):
        """Create a new model"""


        self.new = QMessageBox.question(self.ui, "Message", "New model?",
                                        QMessageBox.Yes | QMessageBox.No,QMessageBox.
        if self.new == QMessageBox.Yes:


            self.newSave = QMessageBox.question(self.ui, "Message", "Do you want to sa
```

```
                                    QMessageBox.Yes | QMessageBox.No,QMessageBox.No)

            if self.newSave == QMessageBox.Yes:
                self.onActionSave()



            vis = ps.Visualisation(self.inputData, self.outputData)
            vis.closeAll()

            self.initModel()
            self.updateModel()
            self.filename = None


    def onActionSave(self):
        """Save model"""

        self.updateModel()

        if self.filename is None :
            self.filename, _ = QFileDialog.getSaveFileName(self.ui,
                'Save model', '', 'Modell filer (*.json)')

            #If the user selects cancel in the save window, self.filename
            #will contain self.filename = '', i.e nothing.
            if self.filename == '':
                self.filename = None



        if  self.filename is not None:
            QMessageBox.information(self.ui,"Message", "The model has been saved")
            self.inputData.save(self.filename)




    def onActionSaveAs(self):
        """Save a new model"""

        self.updateModel()


        self.filename, _ = QFileDialog.getSaveFileName(self.ui,
                'Spara modell', '', 'Modell filer (*.json)')

        #If the user selects cancel in the save window, self.filename
        #will contain self.filename = '', i.e nothing.
        if self.filename == '':
            self.filename = None

        if self.filename is not None:
            self.inputData.save(self.filename)

    def onActionExit(self):
        """Close"""

        self.exit = QMessageBox.question(self.ui, "Meddelande", "Exit?",
                                         QMessageBox.Yes | QMessageBox.No,QMessageBox
        if self.exit == QMessageBox.Yes:
```

```
        sys.exit()


def onActionOpen(self):
    """Open a existing file"""


    self.filename, _ = QFileDialog.getOpenFileName(self.ui,
    "Open model", "", "Modell filer (*.json *.jpg *.bmp)")

    if self.filename == '':
        self.filename = None

    if self.filename is not None:
        QMessageBox.information(self.ui, "Message", self.filename)
        self.inputData.load(self.filename)
        self.updateControls()



def onActionExecute(self):
    """Run simulation"""

    #Close all windows
    ps.Visualisation.closeAll(self)


    self.exit = QMessageBox.question(self.ui, "Message", "Run simulation?",
                                     QMessageBox.Yes | QMessageBox.No,QMessageBox.

    # --- Deactivate interface/ui during simulation.

    if self.exit == QMessageBox.Yes:
        self.ui.setEnabled(False)

    # --- Update values from the controls
        self.updateModel()

    # --- Create a solver
        self.solver = ps.Solver(self.inputData, self.outputData)


    # --- Create a thread to run the simulation in, so the interface
    #     wont freeze.

        self.solverThread = SolverThread(self.solver)
        self.solverThread.finished.connect(self.onSolverFinished)
        self.solverThread.start()


def calcDone(self):

    if self.outputData.coords == None:

        QMessageBox.information(self.ui, "Message","Not possible to obtain results



def onShowGeometry(self):
    """Show geometry window"""
    self.calcDone()
```

```
        vis = ps.Visualisation(self.inputData, self.outputData)
        if self.outputData.coords != None:
            vis.showGeometry()

    def onShowMesh(self):
        """Show mesh window"""
        self.calcDone()

        vis = ps.Visualisation(self.inputData, self.outputData)

        if self.outputData.coords != None:
            vis.showMesh()

    def onShowNodalValues(self):
        """Show nodal values window"""
        self.calcDone()
        vis = ps.Visualisation(self.inputData, self.outputData)

        if self.outputData.coords != None:
            vis.showNodalDisplacement()


    def onShowElementValues(self):
        """Show element values window"""
        self.calcDone()
        vis = ps.Visualisation(self.inputData, self.outputData)

        if self.outputData.coords != None:
            vis.showElementValues()

    def onValueChanged(self):
        self.ui.labelSlider.setText(str(self.ui.ElemSlider.value()/1000))

    def onWriteReport(self):
        """ Write report to a text file"""

        self.calcDone()
        if self.outputData.coords != None:

            self.writeReport = QMessageBox.question(self.ui, "Message", "Do you want t
                                    QMessageBox.Yes | QMessageBox.No,QMessageBox.No)

            if self.writeReport == QMessageBox.Yes:

                self.filenameReport, _ = QFileDialog.getSaveFileName(self.ui,
                    'Save model', '', 'Model filer (*.txt)')

                if self.filenameReport!= "":
                    self.report.writeToFile(self.filenameReport)


    def onExecuteParamStudy(self):


        """Execute parameter study"""

        # --- collect graphical interface

        self.inputData.paramA = self.ui.aRadioButton.isChecked()
        self.inputData.paramB = self.ui.bRadioButton.isChecked()
```

```
        if self.inputData.paramA:
            self.inputData.aStart = float(self.ui.aEdit.text())
            self.inputData.aEnd = float(self.ui.aEndEdit.text())
            self.inputData.paramFilename = "paramStudy"
            self.inputData.paramSteps = int(self.ui.paramSpinBox.value())

            # --- Update values from control
            self.updateModel()

            # --- start a solver thread,

            self.solverThread = SolverThread(self.solver, paramStudy = True)
            self.solverThread.finished.connect(self.onSolverFinished)
            self.solverThread.start()

        elif self.inputData.paramB:
            self.inputData.bStart = float(self.ui.bEdit.text())
            self.inputData.bEnd = float(self.ui.bEndEdit.text())
            self.inputData.paramFilename = "paramStudy"
            self.inputData.paramSteps = int(self.ui.paramSpinBox.value())

            # --- Update values from control
            self.updateModel()

            # --- start a solver thread,

            self.solverThread = SolverThread(self.solver, paramStudy = True)
            self.solverThread.finished.connect(self.onSolverFinished)
            self.solverThread.start()
        else:
            QMessageBox.information(self.ui,"Message" , "Please ensure that parameter


    def updateControls(self):
        """Update the controls from the window"""

        self.ui.wEdit.setText(str(self.inputData.w))

        self.ui.hEdit.setText(str(self.inputData.h))

        self.ui.aEdit.setText(str(self.inputData.a))

        self.ui.bEdit.setText(str(self.inputData.b))

        self.ui.tEdit.setText(str(self.inputData.t))

        self.ui.eEdit.setText(str(self.inputData.E))

        self.ui.vEdit.setText(str(self.inputData.v))

        self.ui.qEdit.setText(str(self.inputData.q))

        self.ui.ElemSlider.setValue(int(self.inputData.elSizeFactor))

        self.ui.aEndEdit.setText(str(self.inputData.aEnd))

        self.ui.bEndEdit.setText(str(self.inputData.bEnd))
```

```
    def updateModel(self):
        """Collect values from the window"""

        self.inputData.w = float(self.ui.wEdit.text())

        self.inputData.h = float(self.ui.hEdit.text())

        self.inputData.a = float(self.ui.aEdit.text())

        self.inputData.b = float(self.ui.bEdit.text())

        self.inputData.t = float(self.ui.tEdit.text())

        self.inputData.E = float(self.ui.eEdit.text())

        self.inputData.v = float(self.ui.vEdit.text())

        self.inputData.q = float(self.ui.qEdit.text())

        self.inputData.elSizeFactor = float(self.ui.ElemSlider.value())


    def initModel(self):
        """ Create neccesary objects to indata, outdata and solution."""

        self.ui.reportEdit.clear()
        self.inputData = ps.InputData()
        self.outputData = ps.OutputData()

        self.solver = ps.Solver(self.inputData, self.outputData)


        self.inputData.ptype = 1
        self.inputData.magnfac = 1000

        self.inputData.w = 0.3
        self.inputData.h = 0.1
        self.inputData.a = 0.05
        self.inputData.b = 0.025
        self.inputData.t = 0.15
        self.inputData.E = 2.08e10
        self.inputData.v = 0.2
        self.inputData.q = 100e3
        self.inputData.elSizeFactor = 50
        self.inputData.aStart = self.inputData.a
        self.inputData.bStart = self.inputData.b
        self.inputData.aEnd = self.inputData.a
        self.inputData.bEnd = self.inputData.b
        self.inputData.paramSteps = 1
        self.inputData.paramFileName = None
        self.inputData.paramA = False
        self.inputData.paramB = False


        self.updateControls()

if __name__ == '__main__':

    # --- create application

    app = QApplication(sys.argv)
```

```
    # --- Show main window

    widget = MainWindow()
    widget.show()
    widget.initModel()
    widget.updateModel()
    # --- Start loop
    sys.exit(app.exec_())
```

7.2 PlaneStress.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Mar 22 16:01:51 2017

@author: kajsa
"""

import numpy as np
import json
import calfem.core as cfc
import calfem.geometry as cfg  # <-- Geometry
import calfem.mesh as cfm      # <-- Mesh generating
import calfem.vis as cfv       # <-- Visualisation
import calfem.utils as cfu     # <-- mixed routines
import pyvtk as vtk          # Paraview module


class InputData(object):
 #   """Class to define inputdata to the model."""

    def __init__(self):

        # Version
        self.version = 1


    def geometry(self):
        """Create a geometry instance based on the
        previously defined parameters"""

        #Create the geometry instans

        g = cfg.Geometry()

        w = self.w
        h = self.h
        a = self.a
        b = self.b



        #Create points for the geometry

        g.point([0, 0])              # 0
        g.point([(w-a)/2, 0])        # 1
        g.point([(w-a)/2, b])        # 2
        g.point([w/2+a/2, b])        # 3
        g.point([w/2+a/2, 0])        # 4
        g.point([w, 0])              # 5
```

```
        g.point([w, h])             # 6
        g.point([w/2+a/2, h])       # 7
        g.point([w/2+a/2, h-b])     # 8
        g.point([(w-a)/2, h-b])     # 9
        g.point([(w-a)/2, h])       # 10
        g.point([0, h])             # 11


        # Link the points with splines
        # Use a marker to define splines with a load or boundary condition

        g.spline([0, 1])            # 0
        g.spline([1, 2])            # 1
        g.spline([2, 3])            # 2
        g.spline([3, 4])            # 3
        g.spline([4, 5])            # 4
        g.spline([5, 6], marker=10) # 5
        g.spline([6, 7])            # 6
        g.spline([7, 8])            # 7
        g.spline([8, 9])            # 8
        g.spline([9, 10])           # 9
        g.spline([10, 11])          # 10
        g.spline([11, 0],marker=20) # 11


        # Create the surface
        g.surface([0,1,2,3,4,5,6,7,8,9,10,11])

        # Return the geometry
        return g

    def save(self, filename):
        """Save inputdata to file."""

        inputData = {}
        inputData["version"] = self.version
        inputData["t"] = self.t
        inputData["E"] = self.E
        inputData["v"] = self.v
        inputData["w"] = self.w
        inputData["h"] = self.h
        inputData["a"] = self.a
        inputData["b"] = self.b
        inputData["q"] = self.q
        inputData["elSizeFactor"] = self.elSizeFactor
        inputData["aStart"] = self.aStart
        inputData["bStart"] = self.bStart
        inputData["aEnd"] = self.aEnd
        inputData["bEnd"] = self.bEnd
        inputData["Number of steps"] = self.paramSteps

        ofile = open(filename, "w")
        json.dump(inputData, ofile, sort_keys = True, indent = 4)
        ofile.close()

    def load(self, filename):
        """Read inputdata from file."""

        ifile = open(filename, "r")
        inputData = json.load(ifile)
        ifile.close()
```

```
        self.version = inputData["version"]
        self.t = inputData["t"]
        self.E = inputData["E"]
        self.v = inputData["v"]
        self.w = inputData["w"]
        self.h = inputData["h"]
        self.a = inputData["a"]
        self.b = inputData["b"]
        self.q = inputData["q"]
        self.elSizeFactor = inputData["elSizeFactor"]
        self.aStart = inputData["aStart"]
        self.bStart = inputData["bStart"]
        self.aEnd = inputData["aEnd"]
        self.bEnd = inputData["bEnd"]
        self.paramSteps = inputData["Number of steps"]



class Solver(object):
    """Class to manage the solution of the model"""
    def __init__(self, inputData, outputData):
        self.inputData = inputData
        self.outputData = outputData

    def execute(self):

        # Transfer to model parameters
        E = self.inputData.E
        t = self.inputData.t
        v = self.inputData.v
        q = self.inputData.q
        h = self.inputData.h
        ptype = self.inputData.ptype
        ep=[ptype,t]
        elSizeFactor = self.inputData.elSizeFactor/1000



       # Call on inputData for geometry
        geometry = self.inputData.geometry()

        # Mesh generating
        elType = 3      # Four node element
        dofsPerNode= 2  # stress-strain --> 2 degrees of freedom

        meshGen = cfm.GmshMeshGenerator(geometry)
        meshGen.elSizeFactor = elSizeFactor
        meshGen.elType = elType
        meshGen.dofsPerNode = dofsPerNode

        # Calculating mesh properties
        coords, edof, dofs, bdofs, elementmarkers = meshGen.create()
        nDofs = np.size(dofs) # Number of dofs
        nelm = edof.shape[0] # Number of elements

        # Extract ex and ey from coord-matrix
        ex,ey = cfc.coordxtr(edof,coords,dofs)

        # Initiating global matrix
        K=np.matrix(np.zeros((nDofs,nDofs)))
```

```
f=np.matrix(np.zeros((nDofs,1)))
stress=np.matrix(np.zeros([nelm,3]))

bc = np.array([],"i")
bcVal = np.array([],"i")

# Constitutive matrix
D=cfc.hooke(ptype,E,v)

# Calculating Stiffness matrix
for elx, ely, eltopo in zip(ex,ey,edof):
    # Element stiffness matrix
    Ke=cfc.planqe(elx,ely,ep,D)
    # Assemble stiffness matrix into global
    cfc.assem(eltopo,K,Ke)

# Force Vector
cfu.applyforcetotal(bdofs,f,10,q*h,dimension=1)

# Apply boundary conditions
bc, bcVal = cfu.applybc(bdofs,bc,bcVal,20,value=0.0)

# Solve the equation system
a,r = cfc.solveq(K,f,bc,bcVal)

# Extracting elemental displacement
ed=cfc.extractEldisp(edof,a)

# Calculate the stress and Von Mises Stress
i=0
vonMises=[]
stress_1 = []
stress_2 = []


for elx, ely, eld in zip(ex,ey,ed):
    [stress[i],_]=cfc.planqs(elx,ely,ep,D,eld)
    vonMises.append(np.sqrt(pow(stress[i,0],2)-stress[i,0]*stress[i,1]+pow(str


    w,v = np.linalg.eig(np.array([[stress[i,0],stress[i,2],0],[stress[i,2],str

    stress_1.append(v[0].tolist())
    stress_2.append(v[1].tolist())


    i=i+1



# Extracting the maximum von mises stress
MaxStress = np.max(vonMises)

# Extracting the minumum von mises stress
MinStress = np.min(vonMises)

# Extracting the max displacement node
MaxDisp = np.max(np.abs(a))

# Extracting the min displacement node
MinDisp = np.min(np.abs(a))
```

```
        # Outputdata
        self.outputData.coords = coords
        self.outputData.edof = edof
        self.outputData.dofs = dofs
        self.outputData.a = a
        self.outputData.r = r
        self.outputData.ed = ed
        self.outputData.stress = stress
        self.outputData.geometry = geometry
        self.outputData.elType = elType
        self.outputData.dofsPerNode = dofsPerNode
        self.outputData.vonMises = vonMises
        self.outputData.MaxStress = MaxStress
        self.outputData.MinStress = MinStress

        self.outputData.topo = meshGen.topo
        self.outputData.stress1 = stress_1
        self.outputData.stress2 = stress_2
        self.outputData.MaxDisp = MaxDisp
        self.outputData.MinDisp = MinDisp

    def executeParamStudy(self):
        """Execute parameter study"""
        # -- Store previous values of a and b
        old_a = self.inputData.a
        old_b = self.inputData.b

        i = 1

        if self.inputData.paramA:

            # --- Create values to perform simulation

            aRange = np.linspace(self.inputData.aStart, self.inputData.aEnd,
                self.inputData.paramSteps)

            # --- Begin parameterstudy
            i = 1
            for a in aRange:
                print("Executing for a = %g..." % a)

                # --- set the desired parameter in the InputData-instance
                self.inputData.a=a

                # --- Run simulation
                self.execute()


                # --- Export vtk-file
                self.exportVtk("paramStudy_0"+str(i)+".vtk")
                i+= 1

        elif self.inputData.paramB:
                    # --- Create values to perform simulation

            bRange = np.linspace(self.inputData.bStart, self.inputData.bEnd,
                self.inputData.paramSteps)

            # --- Begin parameterstudy
```

```
            i = 1
            for b in bRange:
                print("Executing for b = %g..." % b)

                # --- set the desired parameter in the InputData-instance
                self.inputData.b=b

                # --- Run simulation
                self.execute()

                # --- Export vtk-file
                self.exportVtk("paramStudy_0"+str(i)+".vtk")
                i+= 1


        # --- Reset original values

        self.inputData.a = old_a
        self.inputData.b = old_b


    def exportVtk(self, filename):
        """Export results to VTK"""

        print("Exporting results to %s." % filename)
        # ---  Create points and polygon defined from the mesh

        points = self.outputData.coords.tolist()


        polygons = (self.outputData.topo-1).tolist()


        # --- Results from the simulation is created in separable objects. Points in
        # ---  elementdata in vtk.CellData.
        cellData = vtk.CellData(vtk.Scalars(self.outputData.vonMises, name="mises") ,

        # --- Create a structure for the elementmesh.

        structure = vtk.PolyData(points = points, polygons = polygons)

        # --- Store everything in a vtk.VtkData instance

        vtkData = vtk.VtkData(structure, cellData)

        # --- Save to a file

        vtkData.tofile(filename, "ascii")

class OutputData(object):
    """Class to store the results from the simulation"""

    def __init__(self):
        self.a = None # none creates variables containing nothing
        self.r = None
        self.ed = None
        self.stress = None
        self.vonMises = None
        self.MaxStress = None
        self.MinStress = None
        self.MaxDisp = None
```

```
        self.MinDisp = None


        self.geometry = None
        self.coords = None
        self.edof = None
        self.dofsPerNode = None
        self.elType = None
        self.dofs = None



class Report(object):
    """Class for presentation of inputdata and outpdata in a report "structure" """
    def __init__(self, inputData, outputData):
        self.inputData = inputData
        self.outputData = outputData
        self.report = ""

    def clear(self):
        self.report = ""

    def addText(self, text=""):
        self.report+=str(text)+"\n"

    def __str__(self):
        self.clear()
        self.addText()
        self.addText("-------------- Model input ---------------------------------")
        self.addText()


        self.addText("Parameter Study, a start:")
        self.addText()
        self.addText(self.inputData.aStart)
        self.addText()

        self.addText("Parameter Study, a end:")
        self.addText()
        self.addText(self.inputData.aEnd)
        self.addText()

        self.addText("Parameter Study, b start:")
        self.addText()
        self.addText(self.inputData.bStart)
        self.addText()

        self.addText("Parameter Study, b end:")
        self.addText()
        self.addText(self.inputData.bEnd)
        self.addText()

        self.addText("Parameter Study, parameter step:")
        self.addText()
        self.addText(self.inputData.paramSteps)
        self.addText()

        self.addText("Coordinates:")
        self.addText()
        self.addText(self.outputData.coords)
        self.addText()
```

```
        self.addText("Element degree of freedom,edof:")
        self.addText()
        self.addText(self.outputData.edof)
        self.addText()

        self.addText("Degree of freedom, dof:")
        self.addText()
        self.addText(self.outputData.dofs)
        self.addText()


        self.addText()
        self.addText("-------------- Model output ---------------------------------"
        self.addText()

        self.addText("The size of the magnification factor used for the visual results
        self.addText()
        self.addText(self.inputData.magnfac)
        self.addText()


        self.addText("Nodal displacements, elementwise:")
        self.addText()
        self.addText(self.outputData.ed)
        self.addText()


        self.addText("Max Von Mises stress (Pa):")
        self.addText()
        self.addText(self.outputData.MaxStress)
        self.addText()

        self.addText("Min Von Mises stress (Pa):")
        self.addText()
        self.addText(self.outputData.MinStress)
        self.addText()


        self.addText("Max Nodal displacement (m):")
        self.addText()
        self.addText(self.outputData.MaxDisp)
        self.addText()

        self.addText("Min Nodal displacement (m):")
        self.addText()
        self.addText(self.outputData.MinDisp)
        self.addText()


        return self.report

    def writeToFile(self,filename):
        # Write the report to a text-file
        ofile=open(filename,"w")

        for line in self.report:
            ofile.write(line)
        ofile.close()
```

```
class Visualisation(object):
    def __init__(self, inputData, outputData):
        self.inputData = inputData
        self.outputData = outputData

        # --- Variables which stores references to open figures

        self.geomFig = None
        self.meshFig = None
        self.elValueFig = None
        self.nodalDisplacementFig = None

    def showGeometry(self):
        """Show geometry visualization"""

        geometry = self.outputData.geometry

        self.geomFig = cfv.figure(self.geomFig)
        cfv.clf()
        cfv.drawGeometry(geometry, title="Geometry")

    def showMesh(self):
        """Show mesh visualization"""

        coords = self.outputData.coords
        edof = self.outputData.edof
        dofsPerNode = self.outputData.dofsPerNode
        elType = self.outputData.elType

        self.meshFig = cfv.figure(self.meshFig)
        cfv.clf()
        cfv.drawMesh(coords=coords,edof=edof,dofsPerNode=dofsPerNode,elType=elType,fil

    def showNodalDisplacement(self):
        """Show nodal displacement visualization"""

        a = self.outputData.a
        coords = self.outputData.coords
        edof = self.outputData.edof
        dofsPerNode = self.outputData.dofsPerNode
        elType = self.outputData.elType
        magnfac = self.inputData.magnfac

        self.nodalDisplacementFig = cfv.figure(self.nodalDisplacementFig)
        cfv.clf()
        cfv.drawDisplacements(a,coords,edof,dofsPerNode,elType, doDrawUndisplacedMesh=

    def showElementValues(self):
        """Show Von mises visualization"""

        a = self.outputData.a
        coords = self.outputData.coords
        edof = self.outputData.edof
        dofsPerNode = self.outputData.dofsPerNode
        elType = self.outputData.elType
        vonMises = self.outputData.vonMises
        magnfac = self.inputData.magnfac

        self.elValueFig = cfv.figure(self.elValueFig)
        cfv.clf()
        cfv.drawElementValues(vonMises,coords,edof,dofsPerNode,elType,a,doDrawMesh=Tru
```

```
    cfv.colorBar().SetLabel("Effective Stress")


def closeAll(self):
    cfv.closeAll()


def wait(self):
    """This method ensures that the windows are kept updated and will return
    when the last window is closed."""

    cfv.showAndWait()
```